AN APPROACH TO THE PARTITIONING IN SOFTWARE DEVELOPMENT FOR DISTRIBUTED PARALLEL COMPUTING SYSTEMS

By

RUEY-MING YANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1993

To my mother, My wife Mei-Wen, John and Wendy

ACKNOWLEDGMENTS

I would like to express my appreciation and gratitude to my advisor, Dr. Stephen S. Yau, for his careful guidance during this study. I would like to thank Drs. Li-Min Fu, Chung-Yee Lee, Yann-Hang Lee, and Nabil Kamel, my supervisory committee. Without their supervision and counsel, this work would not have been possible.

I would also thank Dr. Irving T. Ho, President of International Integration Systems, Inc., and Dr. Jyh-Shen Ke, Vice President of the Institute for Information Industry in Taiwan, for their help and advice to me to pursue the Ph.D. degree.

I owe much gratitude to my mother and my wife, Mei-Wen, for their continuous encouragement and support for me to complete my Ph.D. degree. I also am grateful to my children, Wendy and John, for their patience and understanding with my frequent absence.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

AN APPROACH TO THE PARTITIONING IN SOFTWARE DEVELOPMENT FOR DISTRIBUTED PARALLEL COMPUTING SYSTEMS

By

Ruey-Ming Yang

May 1993

Chairman: Dr. Stephen S. Yau
Major Department: Computer and Information Sciences

Due to the rapid progress in microelectronic technology, the applications of parallel and distributed computing systems such as process control and telecommunications are becoming feasible. The trend is toward a distributed parallel computing system in which one or more of the sites are made up of parallel processing computing systems. This becomes feasible with the availability of low-cost parallel processing systems, such as transputers and other parallel processing machines.

The software development for distributed parallel software has to take partitioning and allocation into consideration. Partitioning deals with establishing boundaries within the software system to exploit parallelism efficiently. Allocation utilizes the boundaries established in the partitioning stage and assigns them to the processing elements. Partitioning and allocation are important issues in the software development for distributed parallel computing systems to efficiently utilize the computing resources.

This dissertation is devoted to developing partitioning methods which are used in the software development for distributed parallel computing systems. Our approach involves two stages: the design stage and coding stage. The objective of the design stage partitioning is to exploit the potential concurrency among modules and to

avoid nonprofit message traffic under the constraint that some modules cannot reside in the same software component. We partition the module structure graph which is the design specification for the distributed parallel software. Two design stage algorithms are developed, partitioning without module duplication and with module duplication. The applicability of these two algorithms depends on the availability of the computing resources.

Coding stage partitioning is done on the parallel machine site. Execution profile information is used to derive estimates of execution times and data size in the program. The objective of program partitioning is to minimize the execution time of the program. The partitioning method is based on the critical path of the computation graph which represents the program. Nodes with the greatest communication in the critical path are grouped together to reduce the communication overhead. Program code partitioning is done at program compiling time.

Our partitioning approach takes advantage of the available information regarding the potential parallelism at both the design and coding stages. It integrates the partitioning methods with the software development process, and is different from other available partitioning approaches which are done either at the design stage or at the coding stage.

CHAPTER 1
INTRODUCTION

1.1 Scope and Objective

Due to the rapid progress that has been made in microelectronic technology, the applications of parallel and distributed computing systems such as process control and telecommunications are becoming more feasible. The trend is toward a distributed parallel computing system in which one or more of the distributed sites are made up of the parallel machines [Yau92]. A distributed parallel computing system has the following characteristics. (1) The system has a number of sites, each site having certain computing capability which may be provided by a parallel machine. (2) There are communication links among the sites. (3) There are a number of functional components that reside at each site, and the software components of different sites may be synchronous or asynchronous. (4) There are interactions among the functional components that reside at different sites for the purpose of performing some system-wide functions, such as resource sharing and synchronization of execution of the required functional components.

The development of software for such systems has become rather complicated due to the architectural considerations of distributed systems, including the configuration of parallel computing systems at individual sites in the distributed system. Interprocess and intraprocess communications and partitioning and allocation of the software system are all very important issues in such software development. Furthermore, because most of the existing parallel languages are tied to a specific architecture, the

software development approach for each architecture becomes architecture dependent. Although a framework for software development for distributed parallel computing systems is available [Yau92], the current available partitioning approaches for distributed or parallel systems are not applicable for distributed parallel computing systems.

Optimal execution of software in the multiprocessor environment depends on partitioning the software into units for allocation and allocating those units for the shortest execution time possible. In the partitioning stage, we are concerned with establishing boundaries within the software system. We utilize the boundaries established in the partitioning stage to aid us in assigning software components to the processing elements which made up the multiprocessor architecture.

One distinct characteristic for distributed parallel computing systems is that the parallelism can be exploited either between distributed sites or within the parallel machine, so that there exists two-level parallelism in the distributed parallel computing systems. Usually, the communication cost between the distributed sites is higher than the communication cost between processing nodes within a parallel machine, so that the parallelism between two distributed sites needs to be more coarse than the parallelism within the parallel machine. This two-level parallelism characteristic plays an important role in the software partitioning for distributed parallel computing systems.

*Cm** [Schwan85] is an example of distributed parallel computing systems. It uses a hierarchical structure with clusters connected by an inter-cluster bus. Each cluster is like a single shared-bus multiprocessor with processors connected to a local bus. Communication between two processors in the same cluster is significantly cheaper than between two processors in different clusters.

Currently, the approaches of partitioning for distributed and parallel software are done either at the software design stage or during the coding stage. Approaches for software partitioning done at the design stage attempt to be more application oriented. Software partitioning is based on high-level resource directives or program module constraints stated by the application programmer, where each directive expresses allocation preferences or constraints concerning a set of program modules. One advantage of this approach is that it acts as an extension of traditional software design approaches. The software partitioning algorithm determines the system-level concurrency relationships based on information provided by the designer. However, this approach does not completely consider the "communication cost versus computation gain" trade-off because it is difficult to quantify the parameters needed, such as the module performance and the data volume transmitted between modules at the design stage.

Partitioning also can be done at the coding stage. This approach considers the problem without regard for the solution fit into the overall problem of multiprocessor software design. Many of the partitioning and allocation methods rely on the graph theoretic algorithm or other mathematics-based techniques to analyze the program code and to partition the code to minimize the completion time. These techniques are based upon minimizing some variables, which of course must be quantifiable. Thus, most designers attempt to derive a partition or allocation that minimizes the cost function involving computation and communication in the system.

This study attempts to find partitioning methods for distributed parallel computing systems. We will consider partitioning as the integral part of the entire software development process for distributed parallel software. We use a two-stage partitioning approach. The partitioning is done at both the software design stage and the

program coding stage. This approach will take advantage of both design and coding stage partitioning approaches. We believe that a better software partitioning should include both the potential concurrency information provided by the designer at the design stage and analyzing program at the coding stage. This two-stage partitioning approach also matches the characteristics of distributed parallel computing systems in which exists two-level parallelism (parallelism between distributed sites and parallelism within the parallel computer).

In this dissertation, we integrate partitioning and allocation into the software development process for distributed parallel software. We also design stage partitioning algorithms which analyze the concurrency relationship between software modules and group them into basic units for allocation. For coding stage partitioning, we also develop a method to estimate the communication and computation time for the program code and partitioning algorithm based on critical path analysis. We use the existing available algorithms for the allocation both in the design and coding stages.

1.2 Dissertation Outline

This dissertation consists of six chapters. In addition to this introduction, the rest of the dissertation is organized as follows.

Chapter 2 provides background information regarding distributed parallel software partitioning, including the distributed computing system, parallel computers, issues in the software partitioning, and allocation for multiprocessor systems, and we will review the related researches in the partitioning for distributed or parallel environment.

In Chapter 3, we will discuss our proposed software development process for distributed parallel computing systems. This development process will integrate with

our proposed two-stage partitioning and allocation approach. We will use concurrently executable modules for the specification of distributed parallel software. The construct of the concurrently executable module and the interconnection structure for the software using modules are also discussed in each section. Finally, an example using the above design specification method is given.

Chapter 4 includes discussion of design stage partitioning. We develop design stage partitioning algorithms. Based on design specification, design stage algorithms will partition the modules into Distributed Processing Components (DPCs), which will be used as the basic unit for allocation. According to the availability of computing resources, two kinds of algorithms can be used. Partitioning without module duplication algorithm is for the environment with no sufficient resources for module duplication. Partitioning with module duplication algorithm can generate better performance, while needing more computing resources.

In Chapter 5, we describe the coding stage partitioning method. The computation graph is introduced to represent a program at the compile time. Execution profile information for the communication data size between nodes and frequency of function call is used to estimate program execution time and communication overhead. An algorithm is developed to partition the program in each parallel site at program compile time and allocate them to nodes of the parallel machine at program run time.

Chapter 6 includes some concluding remarks and comparisons of our approach with other approaches. Possible future research directions also are discussed in this chapter.

CHAPTER 2
BACKGROUND

In this chapter, we will present an overview of research related to distributed computing systems, parallel computers, and partitioning, allocation.

2.1 Distributed Computing Systems

Over the past decade advances in hardware technology and the desire to automate increasingly complex applications have lead to increased interest in distributed computing environments. Architecturally, a distributed computing system is a system composed of a set of computers (processing nodes) interconnected by a communication network (see Fig. 2.1). The system is said to be loosely coupled because the processing nodes communicate with one another by exchanging messages through the communication network. Actually, some of the processing nodes themselves may be composed of parallel computers or a set of computers, and they may communicate by means of shared memory. But it is the internode communication that is most important to distributed system designers, and this is done by message passing.

Physically, the processors of the network may be locally or geographically dispersed, leading to the terms "local area networks" (LAN) and "wide area networks" (WAN). Especially for WANs, but also for LANs, message passing is a very critical issue from the perspective of reliability and performance. Message passing communication typically involves run-time support from the operating system, for example , for message formatting, and some varying degrees of message routing and flow control by communication network. The effect is that once a message leaves its source,

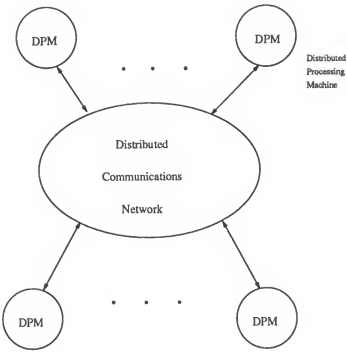


Figure 2.1. Architecture of a Distributed Computing System

it becomes vulnerable to being corrupted or lost, and the transmit time to reach its destination can become long.

The advantages of distributed computing systems mainly are (1) potential for increased performance due to the ability to exploit parallelism, (2) potential for increased reliability due to the redundancy implied by multiple processing components, and (3) potential for natural system growth by adding more processing nodes.

A distributed computing system can be used in two basic ways. The first way, which is called *network computing*, is characterized by a time varying sequence of jobs that arrive at the various processing nodes of the system. These jobs are mainly independent in function, and the goal of the distributed system is to serve primarily as a resource-sharing network. The second form, which is called *cooperative computing*, is characterized by a fixed set of processes that closely cooperate to achieve

some common goal or objective. In this dissertation, our focus will be on using the distributed system for cooperative computing.

2.2 Parallel Computers

Parallel computers, which are the constituents of distributed parallel computing systems, can be classified as being *synchronous, MIMD, and MIMD-based architectural paradigm* according to the taxonomy by Duncan [Duncan90]. Fig. 2.2 is the high-level taxonomy of parallel computer architectures.

Synchronous parallel architectures coordinate concurrent operations in lockstep through global clocks, central control units, or vector unit controllers. Vector computers (e.g. Cray X-MP/4, ETA-10), SIMD machines (e.g., processor array, associative memory architecture), and systolic architecture machines (e.g., Carnegie-Mellon's Warp) belong to this category.

MIMD architectures employ multiple processors that can execute independent instruction streams using local data. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Although software processes executing on MIMD architectures are synchronized by passing messages through an interconnection network or by accessing data in shared memory units, MIMD architectures are asynchronous computers, characterized by decentralized hardware control.

The parallel computers that belong to the category of *MIMD-based architectural paradigm* are based on the same MIMD principles of asynchronous operation and concurrent manipulation of multiple instruction and data streams. However, each of these machines also is based on a distinctive organizing principle as fundamental to its overall design as MIMD characteristics. MIMD/SIMD architectures (e.g., Texas Reconfigurable Array Computer or TRAC), dataflow architectures (e.g., Manchester

Dataflow Computer), reduction architectures (e.g., Newcastle Reduction Machine) and wavefront array architectures fall in the category of MIMD-based architectural paradigms.

Most of the parallel computers in the taxonomy of synchronous and MIMD-based paradigm architecture are targeted to solve specific problems, so that they are for special purpose use, whereas most of the parallel machines in the taxonomy of MIMD are targeted for general purpose use. Parallel computers are the constituents of distributed parallel computing systems. In this dissertation, we will consider only the MIMD machines and exclude synchronous and MIMD-based paradigm machines, due to the generality of MIMD machine applications.

The MIMD parallel machines can be classified as being tightly coupled (shared memory) or loosely coupled (distributed memory). Processors in a tightly coupled machine accomplish interprocessor coordination by providing a global, shared memory that each memory can address. The interprocessor communication cost is lower than for the loosely coupled machines. However, other problems such as data access synchronization and cache coherency must be solved. Coordinating processors with shared variables requires an atomic synchronizing mechanism to prevent one process from accessing a datum before another finishes updating it. Typically, each processor in a shared memory architecture also has a local memory used as a cache. Multiple copies of the same shared memory data, therefore, may exist in various processors' caches at a given time. Maintaining a consistent version of such data is the cache coherency problem, which concerns providing new versions of the cached data to all involved processors whenever a single processor updates its copy. Bus, crossbar, and multistage interconnection networks are the major alternatives for connecting multiple processors to shared memory.

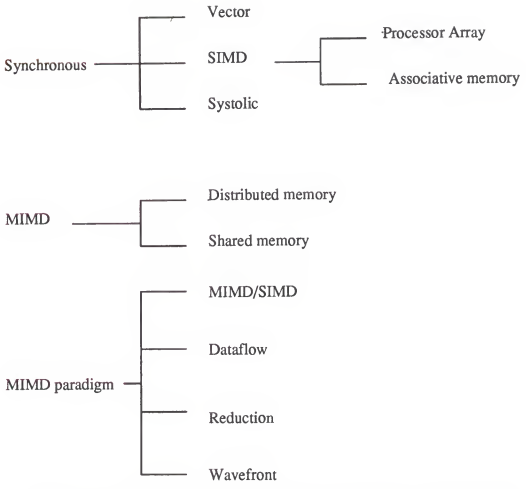


Figure 2.2. High-Level Taxonomy of Parallel Computer Architectures

Loosely coupled MIMD machines connect processing nodes with a processor-to-processor interconnection network. Nodes share data by explicitly passing messages through the interconnection network. The performance and scalability of the multiprocessor is primarily determined by the interconnection network. Various interconnection network topologies have been proposed to support architectural expandabilty and provide efficient performance for parallel programs with differing interprocessor communication patterns. Ring, mesh, tree, hypercube, and reconfigurable topologies have been used for the architecture of loosely coupled MIMD machines.

2.3 Software Partitioning

Optimal execution of software in the multiprocessor environment depends on partitioning the software into modules or tasks and allocating them to the processors for execution. In the partitioning stage, we are concerned with establishing boundaries within the software system. The objective of partitioning must support the design goal, such as minimization of completion time, load balancing, maximization of reliability, or potential for system growth. For example, some software partitioning objectives include

1. Minimizing interprocess communication: Since remote, message passing communication is usually less reliable and more time consuming than local communication, using this objective can help by increasing system reliability and shortening completion time.

2. Exploiting potential concurrency: In other words, if two modules can execute concurrently, we want to locate them in separate processes. On the other hand, if two modules must execute sequentially, then we may locate them in the same process to reduce the interprocess communication cost. This objective may be useful in cases with critical timing requirements.

3. Limiting size of processes: In comparing a partition consisting of many small processes with one consisting of a few large processes, we note that the first will provide more flexibility for purposes of load balance or growth potential.

These partitioning objectives may be used singly or in combination. Actually, we often will need to consider all of them and then make trade-offs among them. For example, suppose we have six modules A, B, C, D, E, and F, with intermodule precedence relations shown in Fig. 2.3. We assume that the execution time for each

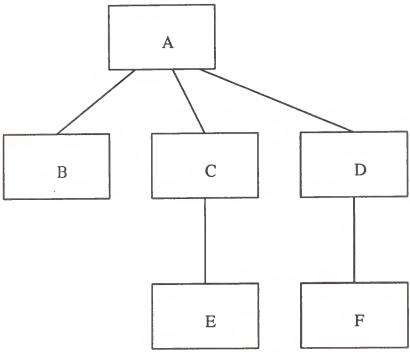


Figure 2.3. Precedence Relations of Modules

module is A:30, B:40, C:30, D:20, E:30, F:70. The execution time can be only an estimate at this stage. Note that module B has the potential for concurrent execution with modules C, D, E, and F; module E has the potential for concurrent execution with modules B, D, and F; and so on.

To exploit potential concurrency, we may partition the set of modules into three processes: p1={A, B}, p2={C, E}, and p3={D, F}. However, if we limit the execution time of any process to 80 units, we may partition as p1={A, D}, p2={B}, p3={C, E}, and p4={F}. Now, suppose that the communication cost between modules B and E is so high that it is not offset by the benefit of exploring their potential concurrent execution (by placing them in separate processes). Then we may partition so that B and E are in the same process. A possible partitioning, again assuming the process execution time limit of 80, is p1={A, D}, p2={B, E}, p3={C}, and p4={F}.

However, if the communication between A and C is more costly than that between A and D, we may let p1={A, C} and p3={D}.

Of course, any partitioning objective must be considered in the face of necessary system constraints such as precedence relations between modules, timing requirements, and limited capacities of computing resources.

In practice, two problems complicate the partitioning activity. First, because partitioning is an earlier design step than allocation, it is difficult to measure the effectiveness of a partition before all modules have been allocated. Second, conflicting partitioning criteria often support the same system requirement. For example, many distributed real time applications have critical response time requirements. To meet these requirements, two common partitioning objectives are minimizing interprocess communication cost and maximizing potential parallelism. To minimize interprocess communication cost, one should partition the system into one process and treat the system as if it were centralized. The interprocess communication cost in this case would be zero. Unfortunately, this strategy does not allow you to exploit any potential parallelism because two computations are eligible to execute in parallel only if they reside in different processes. The inability to exploit parallelism may mean that your design does not meet the response time requirement.

At the other extreme, if you simply partition the system so that each module is a process, all potential parallelism can be exploited. But if these processes are allocated to different processors, all interprocess communication becomes remote. In this case, the heavy communication traffic could degrade system response time, so that the designer would have to take the trade-off between conflicting objectives.

Precedence relations among logical modules cause another type of partitioning trade-off. For example, if module A must terminate before module B can be initiated, the implied precedence relation could motivate the designer to place both

modules in one process. Otherwise, if the processes are assigned to different processors, there would be a cost of remote communication without an associated advantage of exploitable parallelism. Perhaps because of the difficulty of the problem, only a few task partitioning models have been established [Shatz88].

2.4 Task Allocation

Distributed or parallel processing enhances system performance by employing several processors to handle the processing load. We assume that there are a set of modules to be processed, $\{M_1, M_2,..,M_m\}$, and n processors, $\{P_1, P_2,..., P_n\}$. It is convenient to consider a module an indivisible entity, the smallest viable computational unit. In general, the number of modules is much larger than the number of processors. The set of m modules makes up a task; a task is a single processing entity. The processors in this environment communicate among themselves via an interconnection mechanism. Modules may be assigned to different processors. When modules have data to communicate to each other, the processors to which they are assigned must then communicate with each other. Therefore, processors share resources, and communication among processors is essential.

The intermodule communication between any pair of modules is determined by the software design and is a fixed attribute of the modules at the time of allocation. Since the amount of intermodule communication may vary considerably from one pair of modules to another, the way in which modules are allocated to processors can change the overall processing cost dramatically. For example, if two modules, M_i and M_j, have high mutual intermodule communication and they are assigned to different processors, the interprocessor communication load will be high. Total processing cost could be decreased by assigning M_i and M_j to the same processor. On the other hand,

if two modules have little intermodule communication and no precedence relations, assigning them to different processors may well speed up overall processing time.

Task allocation complicates distributed or parallel software design because when you assign m modules onto n processing nodes, there are n^m different assignments. In practice, the situation is even worse because you also must consider data allocation and the potential for both data and process replication. Optimal allocation is a problem of exponential complexity.

The key to task allocation is to establish an allocation model in terms of cost and constraints. The goal is to find a solution that minimizes the cost function under the constraints. Most cost functions are performance oriented, not fault tolerant. Examples of performance oriented cost functions are

1. Total interprocessor communication cost [Chu80]: Interprocessor communication cost occurs when processes residing in different processors must communicate or when a process must access a remote file. Interprocessor communication cost is a function of the amount of data transferred and of network properties such as topology and link capacity.

2. Total execution and interprocessor communication cost [Lo84]: This is the sum of the total computation cost for each process and the total interprocessor communication cost.

3. Completion time [Shen85]: This is the total execution and interprocessor communication cost incurred by that processor whose cost is greater than all other processors.

4. Load balancing [Bannister83]: This measures how evenly spread the workload (process execution time) is across the processors. One reason to seek load

balancing is to maximize system stability. If a system's workload is unbalanced, there may be a processor responsible for substantially more processing than the other processors. In a sense, this processor represents a weak link in the system. Load balancing can be computed as a statistical, normalized coefficient of variation of processor use.

System constraints for allocation include the following:

1. limited memory size and processing capacity of each processor,
2. dependence of some processes on certain processors, requiring the processes be allocated to those processors, and
3. bounds on number of processes on all processors.

The choice of a cost function for a particular system heavily depends on the nature of application and the underlying hardware. For instance, completion time is a critical cost consideration for real time applications, and minimization of total interprocessor communication cost is more important for networks in which processors are geographically dispersed than for local, fully connected networks. For geographically dispersed networks, there is a significant increase in communication time and probability of message loss or corruption.

2.5 Related Research Work

The problems of partitioning and allocation in distributed or parallel processing systems have been studied by many researchers. In the area of partitioning, existing partitioning approaches can be divided into two categories: design partitioning and program partitioning.

2.5.1 Design Partitioning

In design partitioning, Huang [Huang85] developed a modules partitioning model that maps a set of modules into a set of components. Even though the model considers module execution order, an efficient algorithm is still needed to systematically find the desirable solution. Shatz and Yau [Shatz86] proposed a partitioning algorithm for the design of distributed software systems. This method is based on the SADT methodology. The algorithm partitions a set of functional modules in the structure chart graph into distributed processing components. The structure chart is a single-rooted hierarchical graph which has only one control module and so limits the applicability to some distributed applications. Our design stage partitioning without module duplication algorithm is similar to this approach; however, we use different design specifications which allow us to have more than one control module and also better algorithm complexity. Yau and Wiharja [Yau91c] extended the algorithm of Shatz and Yau [Shatz86] with module duplication. Because it also uses the structure chart graph from the SADT method, it inherits the same disadvantage as the previous algorithm. Our design stage partitioning with module duplication algorithm improves the complexity and is applicable to more areas.

2.5.2 Program Partitioning

In the area of program partitioning [Hudak85], a program partition technique based on serial combinators was developed for ALFL language. Serial combinator is a refinement of supercombinator [Peyton87] such that it has no concurrent substructure and is not contained in any larger combinator with the same property. However, serial combinator approaches are not allowed to sacrifice any potential parallelism, resulting in losing the possibility of partitioning the programs into coarse granularity.

SISAL is a high-level dataflow language which has the single assignment property [McGraw85]. Some compile-time partitioning methods have been developed for SISAL programs. In Occamflow [Gaudiot89], the program structure graph and dataflow graph are used as an intermediate form for SISAL programs. Partitioning is based on the intermediate graph, and some methods for scheduling and optimization were developed based on the number of processors and the communication cost. However, this method depends on the architecture of the transputers used. Sarkar et al. [Sarkar88] developed an automatic partitioning compiler for SISAL using IF1 (a dataflow graph) as the intermediate form. The average execution time of the nodes in dataflow graph is estimated by the execution profile information. The granularity of the partitioned program is controlled by a threshold time depending on the number of processors.

A partitioning method was developed based on a scheduling heuristic called DSH (Duplication Scheduling Heuristic) [Kruatrachue88]. It is a compile time partitioning and compile time scheduling method. However, this approach didn't solve the problem of recursive function call in the programs.

A partitioning method based on the concept of clan in the graph theory was developed by McCreary and Gill [McCreary89]. The importance of a clan for partitioning is that the sources and sinks of the clan can been seen as identical in their communication with the rest of the data flow graph. A greedy algorithm is used to find the optimal partition. However, this approach assumes all communication costs are constant and doesn't mention the estimation of execution time.

The data parallel program graph is used as the internal representation of parallel languages by Chatterjee et al. [Chatterjee91]. Compile-time technique, called size and access inference, extracts information about the program variables and statements to partition the graph into regions with different loop sizes. This technique is

used to step up the grain size and reduce storage and synchronization requirements. However, this approach is mainly for program variables of array types.

2.5.3 Allocation

The problem of task allocation in distributed or parallel processing systems has been studied by many researchers, and a taxonomy of the allocation methods was made by Casavant and Kuhl [Casavant88]. Existing task allocation approaches can be divided into categories with the characteristics of static or dynamic, with or without precedence relations.

Stone [Stone77] introduced an approach based on graph theory for the allocation of tasks without precedence relations. It adopts cost function to minimize the total execution and interprocessor communication costs. While it is simple to obtain an optimal solution with this method, it can be applied only to a limited number of processors, which severely limits its applicability.

Chu et al. [Chu80] proposed three task allocation strategies for tasks without precedence relations. One is based on graph theory using a min-cut algorithm. The limitation of this approach is that the complexity of the algorithm becomes intractable when the processor numbers become large. The second strategy uses integer 0-1 programming with the constraints of resources limitation. The previous approaches both use cost function to minimize the total execution and interprocessor communication costs. The third strategy is a heuristic method using fusion to minimize the cost function of minimizing total interprocessor cost. Efe [Efe82] extends the heuristic approach of Chu et al. [Chu80]. It consists of two phases: first, applying a task clustering algorithm to minimize interprocessor communication; second, balancing the load. Shen and Tsai [Shen85] developed a static allocation method to schedule the tasks without precedence relations. The algorithm is based on the graph matching

approach to minimize the cost function of completion time. Towley [Towley86] uses the shortest path technique to solve the problem of finding the optimal assignment over any number of processors of a modular program whose interconnection is in a special tree form called series-parallel graph.

When there are precedence relations among tasks, the goal of the task allocation problem is to reduce the completion time. In general, the task allocation problem for minimizing completion time is known as a multiprocessor scheduling problem, which has been known as an NP problem except in a very few restricted cases. There have been many heuristics proposed to obtain efficient solutions. Among them, list scheduling [Adam74] has been popular because of its simplicity and sub-optimality. In list scheduling, the program is represented as a task precedence graph in which each node represents a task and each edge represents the precedence relation between two tasks. Then the list scheduler assigns a priority to each of the tasks and places tasks in an ordered list according to the priority. When any of the processors is ready to execute, a task with the highest priority is chosen to be executed. Thus, the core of list scheduling is how to determine the priority of each task. The priority can be given to the tasks such as in the critical path [Poly87] or the earliest schedulable tasks [Hwang89].

CHAPTER 3
DESIGN SPECIFICATION OF DISTRIBUTED PARALLEL SOFTWARE

3.1 Development Process for Distributed Parallel Software

We use the framework proposed by Yau et al.[Yau92] and include our two-stage partitioning and allocation approaches to become the software development process for distributed parallel software. Fig. 3.1 shows the major development stages for distributed parallel software.

The process starts at the analysis stage. This stage focuses on analyzing the problem domain and formulating the requirements of the software. Once the analysis has been carried out, a concept model of the software system is produced. This concept model is a very high-level view of the system. There are many methodologies that can be used for this stage. Functional and object-oriented approaches are two popular methods which can be used for system modeling. A functional approach which models the system as a set of the interacting functions and is typified by the data-flow approach. Here, the system is considered to be a set of functional transformations with data flowing from one to another. Another popular method, the object-oriented approach, models the system as a set of interacting objects where the operations allowed on each object are encapsulated with the object itself.

The design process involves describing the system at a number of different levels of abstraction. Effective software design is best accomplished by using a consistent approach to design decomposition. Functional and object-oriented approaches are two major methods. In a functional design, the system is designed from a functional viewpoint, starting with a high-level view and progressively refining this into a more

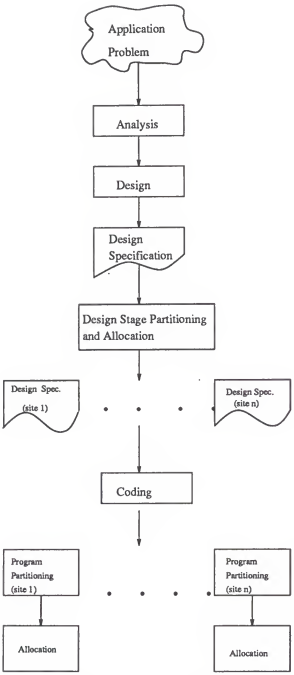


Figure 3.1. Major Development Stages for Distributed Parallel Software

detailed design. Structure charts are a way of representing the hierarchical organization of a system. In an object-oriented design, the system is viewed as a collection of objects rather than as functions, with messages passed from object to object. Object-oriented design is based on the idea of information hiding and is more natural to represent the problem domain. We follow the framework of Yau et al.[Yau92] for distributed parallel software development, in which the object-oriented approach was adopted for the analysis and design stage.

At the analysis and design stage, either using functional or object-oriented approach, we do only the logical design and do not consider the physical configuration of the distributed parallel computing system. The design specification produced also can be implemented in the sequential system. In this dissertation, we develop a design specification for representing the distributed parallel software which will be discussed in the latter parts of this chapter.

The design specification then will be analyzed to find out the potential concurrency and communication between modules. A design stage partitioning algorithm will group the modules into basic units for allocation. The purpose of partitioning is to minimize the communication and maximize the concurrency among modules. The design stage allocation algorithm will schedule each basic unit to different sites of the distributed computing system according to some cost functions such as minimizing interprocessor communication.

After design stage partitioning and allocation, we will get design specification for each distributed site. Because each distributed site is a parallel machine, the design specification will be implemented in the parallel machine.

At the coding stage, the design specification first will be coded into a sequential program in each distributed site. The program will be partitioned into tasks according to the partitioning algorithm which considers the parameters of the parallel machine

such as the number of processors, communication and scheduling overhead. The tasks then will be allocated to processor nodes in the parallel machine. The allocation can then be done at compile time or run time, depending on the scheduling methods.

3.2 Design Specification

The design stage partitioning and allocation tailor the design specification to its hardware environment, such as the number of distributed sites, to exploit the maximum concurrency between modules. In the following sections, we will discuss the design specification for distributed parallel software. The hierarchical organization of a software system is accepted to provide means to keep the complexity of the system manageable and the function of the system understandable. We will specify the overall structure of distributed parallel software by the concept of concurrently executable modules [Weber88] and connect them as a multiple-rooted hierarchical acyclic graph. Each module body is specified as a PROOF object [Yau91] so that the operations in each module are pure functions and will not cause side effect. Parallelism can be exploited between modules or operations within the module body. It matches well with the two-level parallelism characteristic of distributed parallel computing systems: parallelism between distributed sites and within the parallel machine.

3.3 Concurrent Executable Modules

Concurrently executable modules (CEMs) are the units of software that will be used to form distributed parallel software systems. A module specification MOD = (EXP, IMP, BOD) consists of three parts, called Export interface (EXP), Import interface (IMP), and Module body (BOD), respectively. A module is a realization of the module specification, s.t. EXP and IMP remain unchanged and the module body

is realized by programs satisfying the specification in BOD. The general structure of the module is as follows:

MODULE
    EXPORT INTERFACE
    MODULE BODY
    IMPORT INTERFACE
END MODULE

Distributed parallel software is designed as interconnections of modules with matching import and export interface, e.g., the imported modules' export interfaces must match with the importing module's import interface.

3.3.1 Export Interface

The export interface is the visible part needed to be taken to use this module within another module. It allows three different aspects of information hiding:

1. The export interface prevents a user from looking into the internals of a module, i.e., the representation of data and the implementation of operations.

2. The export interface also may protect some of the operations that exist internally from their use by an outside module, i.e., it enables the existence of hidden functions.

3. The export interface also may represent an application-specific hull to the generic module body and import interface part of the module. As such, it may give access to only a subspace of the value space of the encapsulated state data and can be used to enforce application-dependent integrity constraints on the state data.

3.3.2 Import Interface

The import interface contains a number of formal parameters that refer to a set of possible constituent parameterized abstract data types that are each in turn embodied in a different module. This feature of modules is especially suitable to support the stepwise development of software systems. The complete specification of the parameter data types is of concern within the specification of their respective module and may be developed at a later stage in the software development process. In addition, the import interface with its specification of the essential characteristics of the parameter data types leaves room for different actual parameter data types to be reused in the respective module. The concurrency constraint of the subordinate modules also will be specified in the import interface. We will discuss the concurrency constraint later in this chapter.

3.3.3 Module Body

The body of the module defines the construction of the export interface operations in terms of the import interface operations and the operations of the parameter data types. For this purpose the body may contain auxiliary operations, called hidden functions, which do not belong to any other part of the module. The module body follows the structure of the PROOF object [Yau91], each object having its own local data and operations (methods). The operations in PROOF are purely applicative functions, so they will not cause side effects. The guard of an operation is introduced to support the synchronization between concurrent objects. The fine grain parallelism can be exploited at the operation level. For details about the PROOF computation model, see Appendix A of this dissertation.

Modules are meant to be executable units like ordinary programs. Their execution will be initiated from outside, e.g., by users from other modules. Unlike ordinary

programs, however, modules may be executed in a number of different ways through the invocation of a number of different operations that modules export for their use. In addition, some or all the operations in a module's export may be executed concurrently. We, therefore, say a module exports a number of operation names together with a set of parameter names for the eventual concurrent invocation of these operations.

A module is built to encapsulate a number of data objects. The encapsulation will be achieved by associating all, and only those operations with the module that preserves in their execution the consistency of the constructed data objects. The module has an underlying abstract data type and is a self-contained software unit that will be defined independent of each other.

3.4 Concurrency Constraints

Concurrency is a property of the modules. Concurrency constraints will be defined for a module in the import interface. Such concurrency constraints would not be formulated to determine a particular order in the concurrent execution of modules, but rather to determine the set of all permitted executions (i.e., the set of all permitted schedules). It is of no concern in the specification of concurrency constraints how a module will later on be capable of selecting only those schedules that are permitted under the given constraints. That means that a synchronization mechanism that is capable of selecting only permitted schedules will not be part of that specification.

Concurrency constraints will be expressed in terms of the notations from Yau et al.[Yau81]. They enable us to specify which subordinate modules denoted in the import interface of a module must be executed in sequence, which must be executed in a mutual exclusion mode, and which can be executed concurrently. The specification thus determines all the potential orderings of module execution under which the

consistency of the data encapsulated by a module will be preserved. In order to achieve the highest possible degree of concurrency, modules also are equipped with a mechanism to issue invocation requests for operations to subordinate modules in a concurrent fashion.

3.4.1 Sequencing of Module Components

When subordinate modules are executed in a sequence, their control structure can be specified as follows:

SEQ(M_1, M_2, ..., M_n)

where M_1, M_2,...,M_n are n subordinate modules that always are executed from M_1 to M_n in the order M_1, M_2, ..., M_n.

3.4.2 Selection of Module Components

When one of the subordinate modules is selected to be executed according to some test condition, called a selector, the control structure can be specified as follows:

ONE-OF(C; M_1, M_2,..., M_n)

where C is the selector. This control structure is similar to the case statement in the programming languages. The single branching facility, i.e., the if-then-else statement, is a special case of the control structure.

3.4.3 Concurrency of Module Components

When subordinate modules are executed concurrently, regardless of whether or not they involve communication or accessing shared data, these conditions can be

represented as

$$CON(M_1,...,M_n)$$

where M_1 through M_n are component modules that can be invoked concurrently, and components M_1 through M_n are mutually independent of one another.

For example, module M can invoke modules A, B, and C where A and B can execute concurrently, but module C must wait for completion of both A and B. Then, we can express the concurrency constraint in the import interface of module M as the ordering equation SEQ(CON(A,B),C). The SEQ is used to express the sequencing (in left to right order) of modules, and CON is used to express the concurrent execution of modules.

The CON control structure is used to indicate that certain modules can be executed concurrently. However, the problem of possible conflicts among the requests for accessing shared data objects is not considered in this control structure. Because we use the PROOF object in the module body, it will be solved by the operation synchronization mechanism and the data object multimode locking mechanism. Synchronization among objects is achieved by attaching an optional precondition, called guard, to each operation. Each guard is a predicate. The module which invokes the operation is suspended when the attached guard evaluates to "False", and it is resumed when the guard become "True". The guard attached to an operation is defined in such a way that it depends only on the status of the local data and does not depend on the definition of any other operations. In order to ensure the consistency and correctness of local data, a multimode locking mechanism is adopted. Before an operation involving a data object is evaluated, a proper lock for the data object must be obtained. A lock is granted only when it is compatible with other locks granted for

the same object. Such an approach of controlled access to data has the advantage of localizing the manipulation of the data and possible errors associated with the data.

3.5 Interconnection of Modules

The module is the building block for software systems. Based on this, modules will be interconnected to represent the structure of the software system. Distributed parallel software is specified as interconnections of modules with matching import and export interfaces, e.g., the imported module's export interfaces must match with the importing module's import interface. The entire software system will be a partial order (i.e., a hierarchy) on all of the interconnected modules, with one (or more) root module(s) being the last and ultimate importing module, and maybe a number of primitive modules being the last and ultimate exporting modules at the bottom of the hierarchy. In analogy to the specification of the software as a hierarchy, the implementation of the specified system is meant to be a perfect matching hierarchy of runnable modules.

The resulting software system is not a description of one function embodied in a system, but of all functions that may be executed on the distributed parallel computing system. The module hierarchy is not just a functional decomposition of the system but the simultaneous decomposition of operations and data.

In fact, an entire large software system will be seen as a module. This all encompassing "system-module" and all other modules that are not primitive are constructed of other subordinate modules. A module therefore must provide the mechanism for its construction of subordinate modules. A software system constructed out of modules is meant to be a partially ordered set of modules. This is the common denominator for all systems constructed of modules that can be represented by a directed acyclic

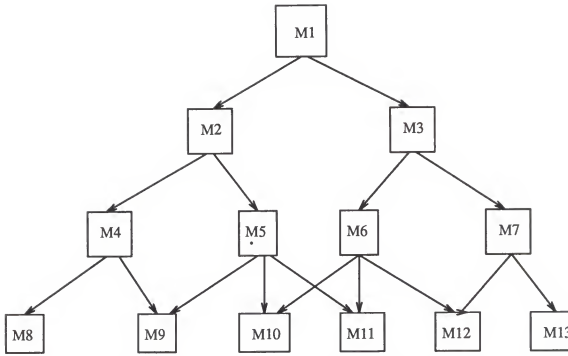


Figure 3.2. Example of Interconnection of Modules

graph, with its nodes representing modules and with the set of edges emanating from a node representing the construction relation.

An example of an acyclic hierarchical graph representing the interconnection of modules is shown in Fig. 3.2. Modules are represented as nodes in the graph. The hierarchical organization of a large software system is accepted to provide a means to keep the complexity of the system manageable and the function of the system understandable.

An invocation of an operation in the top-most module results in the subsequent invocation of the subordinate modules, thus creating an invocation hierarchy in accordance to the given module hierarchy. The invocation of a particular operation

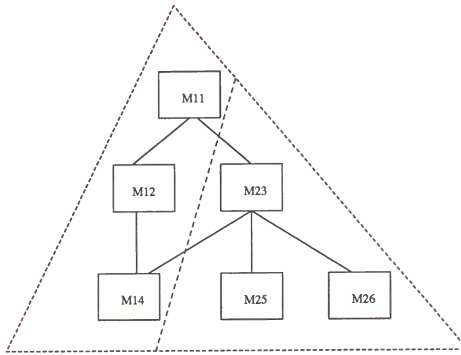


Figure 3.3. Partitioning of Modules

in the top-most module results in one particular invocation structure. The invocation of different operations in the top-most module may result in different invocation hierarchies for the underlying modules.

Invocations of modules are assumed to take place at any time: an operation may be invoked while another module is still in execution, module invocations may take place in strict sequence, and a module may be invoked a second time prior to the termination of the execution of the first operation invocation. For the liberty provided for the invocation of modules, proper ordering for the execution of modules must be enforced in accordance to the specified concurrency constraints.

Invocations of operations at intermediate modules in the module hierarchy may arrive at any time if the module is a subordinate module of many superior modules that are all able to issue invocation requests at arbitrary times. Those modules

must be able to execute in the proper order according to the specified concurrency constraints.

We generally assume each module to reside at one site. This does not prevent us from assuming that different subordinate modules of a given module reside at different sites. More precisely, modules themselves with their constituent export interface, module body, import interface, and common parameter reside at one site and are not subject to distribution. Accordingly, local data and operations of a module are considered to reside at the site of the module. Hence, these also are not subject to distribution.

The distribution of modules of a given distributed parallel software to different sites is the topic of software partitioning and allocation. An example is illustrated in Fig. 3.3. The software system documented consists of the modules M11, M12, and M14 residing at site 1 and of the modules M23, M25, and M26 residing at site 2. We assume there is a centralized control module in the system. The centralized control will be executed by the top-most M11 in the module hierarchy. Systems of this nature frequently are called master/slave systems, where the master residing at one site executes control over the slave residing at another site.

It also is important to note that the model introduced here also provides for distributed and concurrent executions at the various sites. M14 may be the subject of concurrent invocations of operations originating from M12 and M23 that may be executed concurrently if the computational facilities permit it. Our design level partitioning algorithm discussed later in this dissertation is aimed at partitioning the modules and distributing them to different sites in order to exploit more parallelism and avoid unnecessary message traffic for the software.

In the case that two or more modules overlap in some of their subordinate modules, they are said to be cooperate. A multi-rooted hierarchical graph is used to represent

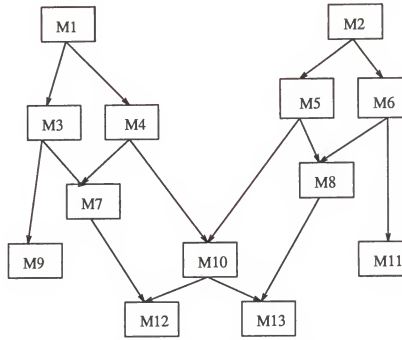


Figure 3.4. Two Rooted Hierarchical Graph

this kind of software. Each root represents an active control module that can invoke the execution of its subordinate modules. A example of a two-rooted hierarchical graph is shown in Fig. 3.4. The software system now is represented by a two-rooted acyclic graph with each of the cooperating systems being represented by one root module with all of its subordinate constituents. Modules M1 and M2 are the two control modules in the root. The systems overlap over modules M10, M11, M12 that are part of both module hierarchies. As in the previous configuration, centralized control is executed by the respective root module over its proper module hierarchy.

3.6 An Example

In this section, we will use the bounded buffer problem to demonstrate the design specification. The Bounded_Buffer is a FIFO queue of limited capability. We can produce the data item and then put it in the buffer, or get the data item from the

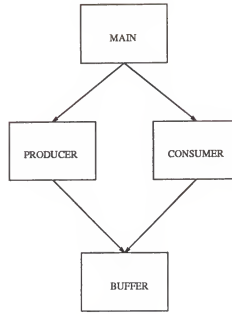


Figure 3.5. Interconnections of Modules for Bound_Buffer Problem

buffer for consumption. There are limitations for the buffer: if the buffer queue is full, you can not put in any more data items; if the queue is empty, you can not get any data items from the buffer. The software system for the Bounded_Buffer problem will consist of four modules. The Main module initiates the execution of the Bounded_Buffer software, the Producer module produces the data item for the buffer, the Consumer module consumes the data item received from the buffer, and the Buffer module encapsulates the content of Bounded_Buffer and the operations that are allowed to be operated on the buffer. These four modules are connected in a hierarchical structure with the Main module as the root. Fig. 3.5 is a representation of the interconnection of the modules for the Bounded_Buffer software.

The body of the Buffer module has two formal parameters: itemtype and size. Itemtype indicates the type of elements to be stored in the buffer, and size specifies the capability of the buffer. There are two operations allowed on the Buffer: *put*

operation, which puts the produced item in the buffer, and *get* operation, which gets the item out of buffer for consumption. The composition clause defines the local data, which represent the content of the buffer, and the counter, which indicates the number of data items stored in the buffer. The export interface of the Buffer module specifies that operations *put* and *get* can be exported to the superior modules. Because the Buffer module is at the leaf of the hierarchical structure, no specification is needed in the import interface. The following is the specification for the Buffer module:

```
module Buffer
export interface   put(item)
                   get()
import interface
begin module body          Buffer(itemtype, size)
                   composition buffer :: store:list(itemtype) × count:int
                   method put item
                          guard(buffer.count < size)
                          expression
                             β[(append_right item), inc] buffer
                   method get
                          guard(buffer.count > 0)
                          expression
                             [ β[tail,dec] buffer, head(buffer.store) ]
end module body
```

The import interface of the Producer module will contain the *put* operation of Buffer module. The export interface of the Producer module will contain the operation *produce*. The module body will specify the operation *produce*. The complete specification for the Producer module is in the following form.

module Producer
export interface *produce*()
put(item)
import interface *put*(item) of Buffer
begin module body Producer
method `produce :: → itemtype`
end module body

The import interface of the Consumer module contains the operation *get* from the Buffer module. The export interface contains the operations *consume* and *get*. The module body section specifies the details of the *consume* operation. The complete specification of the Consumer module is in the following form:

module Consumer
export interface *consume*(item)
get()
import interface *get*() of Buffer
begin module body Consumer
method `consume :: itemtype →`
end module body

The Main module sits at the top of the hierarchical structure of the Bounded_Buffer software and is responsible for the invocation of the subordinate modules. The import interface of the Main module includes the operations of *put*, *produce*, *get*, and *consume*. The concurrency constraint about direct subordinate modules (Producer and Consumer) also is specified in the import interface. The Producer and Consumer modules can be executed concurrently, and the concurrency constraint can be represented as CON(Producer, Consumer). The module body partly specifies the invocation of Producer and Consumer. The complete specification of the Main module is in the following form:

module Main
export interface
import interface *produce*(), *put*(item) of Producer
consume(item), *get*() of Consumer
CON(Producer, Consumer)
begin module body Main
`while True ((` $\mathcal{R}$ `[[Buffer]] (put Buffer produce)),`
`(` $\beta[\mathcal{R}$ `[[Buffer]], consume] (get Buffer)))`
end module body

CHAPTER 4
DESIGN STAGE PARTITIONING

The purpose of design stage partitioning is to exploit potential concurrency and to avoid nonprofitable message traffic between modules. The partitioning algorithm analyzes the design specifications which describe the concurrency constraints and precedence relationships between modules, and then groups the modules into units for allocation. The design stage allocation distributes the units to the distributed sites of a given distributed parallel computing system.

We develop two partitioning algorithms: one is partitioning without module duplication, and another is partitioning with module duplication. When there are enough computing resources, we can duplicate the modules which potentially will be invoked by more than one module simultaneously.

4.1 Partitioning Without Module Duplication

Before we discuss the design stage partitioning approach, we provide the following definitions to be used in the partitioning algorithms.

Definition 4.1.1 A *module structure graph* $G=(N,E,R)$ is a directed graph with the set of nodes N, the set of edges E, and the set of root nodes R, such that

1. Each module M_i in the design specification is represented by one node n_i in N.
2. The edge $(M_i, M_j) \in E$ if module M_i invokes module M_j in the design specification.

3. R is set of roots, $R \subset N$, G has at least one root node. For each root $r \in R$, there are no nodes $n_i \in N$ such that $(n_i, r) \in E$.

Definition 4.1.2 A Distributed Processing Component (DPC) is a set of modules which are collectively to be allocated to the same distributed site in the distributed parallel computing system.

We will present an algorithm used to partition the *module structure graph* without duplicating modules to different distributed sites as follows.

Algorithm 4.1.1 Partitioning the module structure graph into Distributed Processing Components (DPCs).

Input: A directed acyclic module structure graph G = (N, E, R)

Output: A set of Distributed Processing Components (DPCs)

Procedure:

Begin

1. /* For multiple rooted module structure graph */

   if $|R| > 1$ then

   a. Create a pseudo root r_0

   b. Let N = N $\cup$ $\{r_0\}$.

   c. for every $r_i \in$ R do

      connect r_0 and r_i with a pseudo edge (r_0, r_i).

      Let E = E $\cup$ $\{(r_0, r_i)\}$

      endfor

   d. Specify the immediate successors of r_0 that can run concurrently.

      The new module structure graph becomes G = (N, E, { r_0 })

   endif

2. /* Initialization */

   a. for each module i in N do

      Let $C_i = \phi$

      endfor

   b. for each module i in N do

      if the concurrency constraint of module i specifies

      CON(m_1,..,m_k), where $2 \leq$ k, then

      for each j, $1 \leq$ j $\leq$ k, do

      $C_{m_j} = C_{m_j} \cup \{\{m_1,..,m_k\}-\{m_j\}\}$

      endfor

      endif

      endfor

   c. Create an initial DPC $F_0 = \{r_0\}$, where r_0 is the root of G.

   d. Let Q be a queue and Q = ϕ

   e. enqueue (r_0, Q)

3. /* Traversing each edge of the module structure graph */

   while Q $\neq \phi$ do

   a. Let x = dequeue (Q)

   b. for each immediate successor s of node x

      do enqueue (s, Q)

      endfor

      Let F be a DPC such that x is an element of F.

      Let Child = { s | s is the immediate successor of x }

4. /* Checking if s already belongs to some DPC */

   while child $\neq \phi$

   a. Select the module s in Child which involves the largest amount

of communication with module x.

b. Let Child = Child - {s}.

c. if s is an element of some DPC

then go to step 4

endif

5. /* Finding a DPC for s */

if s is an element of C_n for any n, an element of DPC F

then

a. Find a DPC H such that s is not an element of C_n, where {n | n is an element of H }; create a new DPC H = ϕ, if no such H exist.

b. Let H = H $\cup$ {s}

else

c. Let F = F $\cup$ {s}

endif

endwhile

6. Let Father = {z | z is an immediate predecessor of x in G}

7. while Father $\neq \phi$ do

a. Let z be any module $\in$ Father

Father = Father - {z}

b. if z $\in C_x$ then go to step 7.

endif

8. /* Updating the concurrency set of x */

a. Let SIB_x = {y | y is an immediate successor of z and y $\neq$ x and y is specified to be CON with x}

b. Let DES_x = {k | k is a descendent of modules in SIB_x}

c. Let $C_x = C_x \cup SIB_x \cup DES_x \cup C_x$

endwhile

endwhile

End

Step 1 of the algorithm is for the case of a multiple rooted module structure graph. We create a pseudo root r_0 and add the concurrency constraints in the import interface of r_0. Concurrency constraints specify that the original roots of the module structure graph can run concurrently. The pseudo root r_0 is just for the partitioning purpose and does not change the behavior of the software which the module structure graph represents. Step 2 is the initialization step. C_i will be the set of modules which can execute concurrently with module i. Initial C_i can be obtained from the concurrency constraints specified by CON in the module import interface. Due to the hierarchical structure of the software system, the breadth first order is used to traverse the graph. The breadth first traversal is implemented by the data structure of queue. The *enqueue* and *dequeue* operations are used, where the operation enqueue (x, Q) inserts element x at the end of the queue Q and the operation dequeue (Q) deletes the first element of the queue Q. Initially, the root of graph G is inserted in the queue. We also will create the initial DPC F_0 for the root.

While the content of the queue Q is not empty in step 3, we will consider the first module x in the queue and put the immediate successors of x in the queue (which is in the breadth first traversal order). The set of the immediate successors of x is named Child. Steps 4 and 5 will determine each immediate successor of x to which DPC it should belong. We also locate a DPC F where we would like to place these modules together in order to reduce the communication costs involved in their invocation.

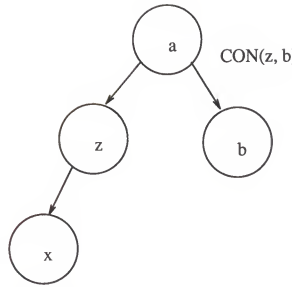


Figure 4.1. Criteria 1 for Finding Potentially Concurrent Modules

We select the module which involves the largest amount of communication with its invoker. This is done in order to minimize the communication cost and is necessary to exploit concurrency.

Steps 7 and 8 determine all the modules which can execute concurrently with module x utilizing the following concepts.

1. If module z invokes module x, then module x can execute concurrently with any modules which can execute concurrently with module z. See the illustration in Fig. 4.1. Module x can execute concurrently with module b.

2. If there exists module y which shares the same immediate predecessor z with module x, and module x is specified to execute concurrently with module y, then module x can execute concurrently with module y and all of module y's descendants. See the illustration in Fig. 4.2. Module x can execute concurrently with module a, b, and c.

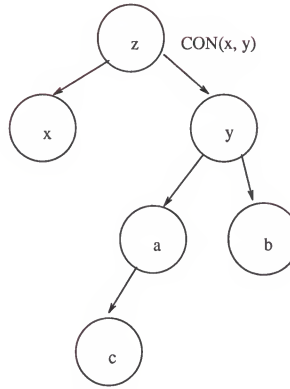


Figure 4.2. Criteria 2 for Finding Potentially Concurrent Modules

If module x is revisited, then for every module z in Father set, we simply union the concurrency set of x with the concurrency set of z. The complexity of this algorithm can be estimated by the following theorem.

Theorem 4.1.1 The algorithm for partitioning the module structure graph into DPCs requires O(MAX(ne, $n^2\log n$)) steps on a graph with n nodes and e edges.

Proof of Theorem 4.1.1 Step 1, which only connects the pseudo root r_0 to original roots of the graph, can be done at constant time. In step 2, it takes at most $O(n^2)$ time to initialize each C_i and constant time to initialize the queue Q. Step 3 traverses the graph in the breadth first order. Because the graph is hierarchical, the total steps for the operation enqueue are equal to $O(e)$, so that the while loop at step 3 will iterate e times. Because the maximum number of DPC is the number of the nodes in the graph, step 4 at most takes n steps with e iterations, so that step 4 requires $O(ne)$ steps.

Step 5 requires that we examine each existing DPC, and for each of these we may need to examine each set C_i, where i is an element of the the given DPC. Because no node is an element of more than one DPC, in the worst case, the amounts to examine each node s in the set C_s for the node under consideration takes $O(n)$ steps. Each search can be done in O(logn) time. Thus step 5 can be done in $O(nlogn)$ time. Note that step 5 is executed once for each node; thus, step 5 contributes a complexity of $O(n^2\log n)$ to the algorithm's complexity.

In step 7, for a node x, the number of its immediate predecessors is no more than n, so that the while loop to find the elements in the CON_x and DES_x is less than n steps in step 8. With e iterations in the algorithm, the total complexity of steps 7 and 8 is $O(ne)$.

The complexity of this algorithm will be O(MAX(ne, $n^2\log n$)). The edges e of a graph may vary from n^2 to n, depending on the density of the edges. If the graph is dense, the complexity is close to O(ne); however, if the graph is sparse, the complexity will be close to O($n^2\log n$). □

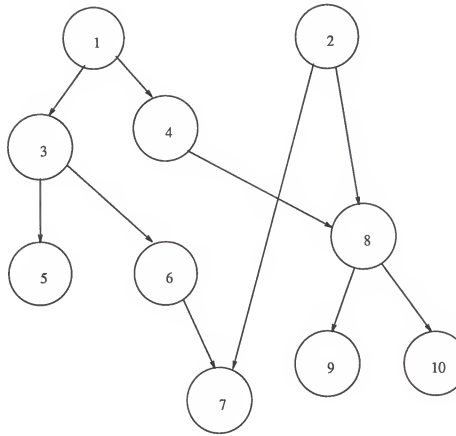
4.1.1 Example

In order to illustrate our algorithm for partitioning the module structure graph into DPCs, we will apply the algorithm to the module structure graph in Fig. 4.3. We assume that the immediate successors of a module are placed from left to right in the order of decreasing communication cost. For simplicity, we name a module with a positive integer. In this example, we use the two-root hierarchical graph which represents a software system having two active modules sitting at the top of a module structure graph. Two-root active modules can concurrently invoke their subordinate modules. The algorithm will create a pseudo root r_0 and connect it with two actual root modules 1 and 2. The edge (r_0, 1) and (r_0, 2) will be with zero communication cost. The pseudo root and edge is used just for the partitioning purpose. At the allocation stage, the pseudo root will not be allocated to the physical distributed site.

The algorithm groups the modules into DPCs according to the criteria that the selected module must not be placed in a DPC which already contains a module with which it can execute concurrently. The partitioning process is done in a breadth first traversal order. The concurrency set for each module also will be found determined. The following shows the final DPCs and the concurrent sets of each node in the graph.

DPC F_0={r_0,1,3,4,5}

DPC F_1={2,7}



Module Relationship:

1. ONE-OF(3,4)
2. CON(7,8)
3. CON(5,6)
8. SEQ(9,10)

Figure 4.3. An Example for Module Structure Graph

DPC F_2={6,8,9,10}

After the partitioning, the original module structure graph will be grouped into several module clusters. In Fig 4.4, we use the dot line to show the boundary of the DPCs. Each DPC will be a task to be allocated to a parallel machine in the allocation stage.

4.2 Partitioning with Module Duplication

At the design stage, the designer sometimes can make a trade-off between performance and the computing resources. Some modules in the distributed parallel software may be invoked concurrently by its predecessors. If we can duplicate them to locate at different distributed sites, it will reduce the communication time of invocation and thus improve the performance. Of course, it will pay the cost of more computing resources like memory. In this section, we will discuss the software partitioning algorithm with module duplication. Before we present the algorithm, we would like to introduce the kind of modules called potential concurrent invocable modules.

Definition 4.2.1 A Potential Concurrent Invocable Module (PCIM) is a module that can be invoked directly or indirectly by other modules simultaneously.

The problem caused by these PCIMs is that they cannot support all the potential accesses to it, and, since one of the goals of our partitioning approach is to exploit the potential concurrency between modules, we propose to duplicate all the PCIMs if the computing resources allowed. We will find out the PCIMs according the concurrency relation between modules. Using the concept stated in the previous algorithm for finding the set of modules which a module can concurrently run with,

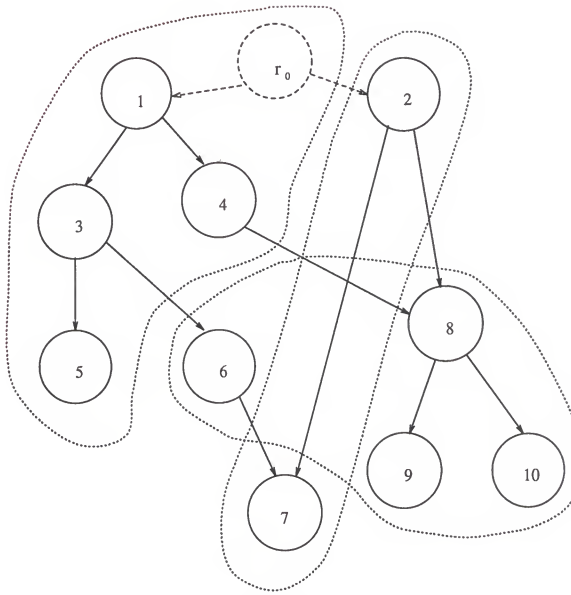


Figure 4.4. Module Structure Graph After Partitioning

1. if module z invokes module x, then module x can execute concurrently with any modules which can execute concurrently with module z, and

2. if there exists module y which shares the same immediate predecessor z with module x, and module x is specified to execute concurrently with module y, then module x can execute concurrently with module y and all of module y's descendants.

A module will potentially concurrently run with itself if it is a PCIM, so that we can find out all the PCIMs by checking the concurrency relations among modules. The approach of partitioning with module duplication will be divided into three major steps:

1. detecting all the PCIMs in the module structure graph G,

2. duplicating the PCIMs detected and keeping the duplicates to the minimal, resulting in the creation of the new module structure graph G^*, and

3. partitioning the graph G^* into DPCs to exploit the potential concurrency between modules and avoid nonprofitable message traffic.

Algorithm 4.2.1 Identifying all PCIMs in the module structure graph G.

Input: A directed acyclic module structure graph G=(N,E,R)

Output: 1. PCIMs in the module structure graph G.

2. C_i for $1 \leq i \leq |N|$, where C_i is the set of modules which can run concurrently with module i.

Procedure:

Begin

1. /* For multiple rooted module structure graph */

if $|R| > 1$ then

a. Create a pseudo root r_0

b. Let N = N ∪ $\{r_0\}$.

c. for every $r_i \in$ R do

connect r_0 and r_i with a pseudo edge (r_0, r_i).

Let E = E ∪ $\{(r_0, r_i)\}$

endfor

d. Specify the immediate successors of r_0 can run concurrently.

The new module structure graph becomes G = (N, E, $\{r_0\}$)

endif

2. /* Initialization */

a. for each module i in N do

Let $C_i = \phi$

endfor

b. Let $PCIM_{set}=\phi$

c. Let Q be a queue and Q = ϕ

d. enqueue (r_0, Q)

3. /* Breadth first traversal on graph G */

while Q ≠ ϕ do

a. Let x = dequeue (Q)

b. for each immediate successor s of node x

do enqueue (s, Q)

endfor

4. Let Father={z | z is an immediate predecessor of x in G}

5. /* Checking all immediate predecessors of module x */

while Father ≠ ϕ do

a. Let z be any module ∈ Father

Father = Father - {z}

b. if z ∈ C_x then go to step 5.

endif

6. /* Updating the concurrency set of module x */

a. Let SIB_x = {y | y is an immediate successor of z and

y ≠ x and y is specified to be CON with x}

b. Let DES_x = {k | k is a descendent of modules in SIB_x}

c. Let $C_x = C_x \cup SIB_x \cup DES_x \cup C_x$

endwhile

endwhile

7. /* Finding out all PCIMs */

for each node i ∈ N do

a. if node i ∈ C_i then

Let $PCIM_{set} = PCIM_{set} \cup \{i\}$

endif

endfor

End

Step 1 of the algorithm is used for the case of a multiple rooted module structure graph by creating a pseudo root r_0. Step 2 is for initializing C_i, $PCIM_{set}$ and queue Q. Step 3 begins to traverse the module structure graph in a breadth first order using queue. In step 4, we find out the immediate predecessors of x. Step 5 considers an immediate predecessor of x at a time. In step 6, we can determine all the modules which can execute concurrently with module x through (1) the concurrency relations

among x and its siblings, or (2) the set of modules which can execute concurrently with its immediate predecessors. Step 7 determines all the PCIMs by examining which module potentially can execute with itself.

Theorem 4.2.1 The algorithm to identify the Potential Concurrent Invocable Modules (PCIMs) requires $O(ne)$ steps on the module structure graph with n nodes and e edges.

Proof of Theorem 4.2.1 Step 1, which only connects the pseudo root r_0 to original roots of the graph, can be done at constant time. Step 2 is the initialization for C_i, $i \in N$, the set of modules which can run concurrently with module i. It will take n steps. Other operations in step 2 take constant time. Step 3 traverses the graph in breath first order. The total steps for enqueue are equal to the edges of the graph, so that the while loop of step 3 will iterate e times. Step 4 takes constant time. In step 5, for a node x, the number of its immediate predecessors is no more than n, so that the while loop to find the elements in the CON_x and DES_x is less than n steps in step 6. With e iterations, the total complexity of step 5 and 6 is $O(ne)$. In step 7, it takes n iterations to check each module if it is a PCIM. Each search takes at most n steps, as the complexity of step 7 is $O(n^2)$. The total complexity of this algorithm is dominated by steps 5 and 6; it will take $O(ne)$ steps. □

The second step for the partitioning is to duplicate the PCIMs detected. If a PCIM has only one immediate predecessor, nothing is to be duplicated. Otherwise, one simple way of duplication is to copy a PCIM to each of its immediate predecessor. However, in some cases it will cause unnecessary duplications, for example, in the case that a PCIM has more than two immediate predecessors but there are only two concurrently executable paths leading to it. Obviously, there are other paths leading

to that PCIM which are not concurrently executable with the mentioned two concurrently executable paths. In this situation, the PCIM should not be duplicated to the nodes which are led from the nonconcurrently executable paths. The duplication algorithm has to determine the minimum duplications for each PCIM.

The number of PCIM duplications will be determined by the elements of the concurrency set in its immediate predecessors. The immediate predecessors of each PCIM will be divided into groups. The modules in the same group cannot run concurrently with each other. Each group will have a duplication of PCIM.

Algorithm 4.2.2 Duplicating PCIMs in the module structure graph G.

Input: 1. A directed acyclic module structure graph G
2. PCIMs in the module structure graph G.
3. C_i for $1 \leq i \leq |N|$, where C_i is the set of modules which can run concurrently with module i.

Output: 1. A new module structure graph G^* after PCIMs duplication
2. C_i for $1 \leq i \leq |N^*|$, where C_i is the set of modules which can run concurrently with module i, in the module structure graph G^*.

Procedure:

Begin

1. /*Duplicating each element in $PCIM_{set}$ */
   while $PCIM_{set} \neq \phi$
2. /* Selecting a PCIM for duplication */
   a. Select node x $\in PCIM_{set}$ which has the highest level number.
      (on conflict, pick any one)
   b. Let $PCIM_{set} = PCIM_{set}$ - {x}
   c. Let Father = {y | y is an immediate predecessor of x $\in$ G}

d. if $|Father| = 1$

then goto step 1

endif

e. Let groupnumber = 1

f. Let $G_x[1] = \phi$

3. /* Dividing the immediate predecessors of module x into groups */

while Father $\neq \phi$

a. Select an y $\in$ Father

Let Father = Father - {y}

b. for i = 1 to groupnumber do

if $\nexists$ z, z $\in G_x$[i] and y $\in C_z$

then

G_x[i] = G_x[i] $\cup$ {y}

goto step 3.

endif

endfor

c. Let groupnumber = groupnumber + 1

d. Create new group G_x[groupnumber]

Let G_x[groupnumber] = {y}

endwhile

4. /* Duplicating a subgraph for each group */

a. Duplicate subgraph G' rooted at x, (groupnumber – 1) times

Call each copy of the subgraphs $X_1,\ldots,X_{groupnumber-1}$

b. for every z node in the (groupnumber – 1) subgraph do

name node z distinctly

Let $C_z = \phi$

endfor

c. Let the root of each subgraph X_i called x_i,

$1 \leq i \leq$ (groupnumber–1)

5. /* Updating the module relationship in the new graph */

a. Let the module relationship of nodes in subgraph (X_i–x_i),

$1 \leq i \leq$ (groupnumber–1), be the same as relations

specified for subgraph (G'–x)

b. for i = 1 to (groupnumber – 1) do

for every node s $\in G_x$[i] do

Delete the edge (s, x)

Construct a edge (s, x_i)

Let Sibling = {y | y is an immediate successor

of s and y $\neq x_i$}

Let module relationship between x_i and nodes in sibling

be the same relationship between x and nodes in sibling.

endfor

endfor

endwhile

6. Using the previous algorithm 4.2.1 to determine C_i

for $1 \leq i \leq |N^*|$, where C_i is the set of modules which can run

concurrently with module i.

End

Step 1 repeats $|PCIM_{set}|$ iterations to duplicate each module in the $PCIM_{set}$. The algorithm will start from the PCIM with the highest level number in the graph.

The level number of a node in the graph is defined as Level-no(x) = max{Level-no(y | y is an immediate predecessor of x)} + 1. This will guarantee that when duplicating a PCIM, all the higher level PCIMs are already duplicated.

Step 2 selects the PCIM to be duplicated according to the level number and checks if there are more than one immediate predecessor for the PCIM. If it has more than one immediate predecessor, we will cluster them into several groups. Each group will share one PCIM copy. The criteria for the modules in the same group is that they will not concurrently invoke the PCIM to be duplicated. Steps 2e and 2f are the initialization for groupnumber and the set of modules in the first group.

In step 3, we will check each immediate predecessor y of x to see if it can run concurrently to invoke x with any module in the existing group. Step 3b selects the group among the groups created that does not contain any module that can execute concurrently with y, then adds y to be a member of that group. If there does not exist any group for y, steps 3e and 3d create a new group for y.

We will duplicate a PCIM for each group of x's immediate predecessors in step 4. After duplication, distinct names are given to all the new nodes created. In step 5, we will add the duplicated nodes to graph G and update the module relationship for those nodes in the newly created graph G^*. Step 6 will use algorithm 4.2.1 to determine the concurrency set for each module in the newly created graph G^*

Theorem 4.2.2 The algorithm to duplicate PCIMs for the module structure graph G requires $O(n^3)$ steps, where n is the number of nodes and e is the number of edges in G.

Proof of Theorem 4.2.2 The number of $|PCIM_{set}|$ is no more than n, so that the while loop in step 1 requires n iteration at the worst case. The selection in step 2a takes at most $\log n$ steps using binary search. Other initialization operations in step

2 take only constant time. The total predecessors of all PCIMs is no more than e, so that the while loop in step 3 is no more than e iterations. The groupnumber of each of PCIM's immediate predecessors is no more than n. The complexity of step 3 is $O(ne)$. Step 4a takes $O(n^2)$ and step 4b needs $O(n^3)$ at the worst case. In step 5b, it will take at most n steps with n iterations. The complexity of step 5 is $O(n^2)$ steps. Step 6 uses algorithm 4.2.1 to determine the concurrency set of each module in the newly created graph; it will take $O(ne)$ steps. The complexity of this algorithm then is dominated by step 4; it takes $O(n^3)$ steps. □

Theorem 4.2.3 In the algorithm to duplicate PCIMs for the module structure graph G, the number of duplicates is minimal.

Proof of Theorem 4.2.3 For each PCIM, x, being considered for duplication to its immediate predecessors, step 3 will group the nodes in the set Father such that nodes in the same group will share the same copy of the PCIM. When node i is being considered to be grouped into a group $G_x[j]$, step 3b checks if node i can execute concurrently with any node in $G_x[j]$. Obviously, nodes that can execute concurrently cannot share the same copy of PCIM. Step 3d creates a new group for node i only if node i cannot share the same copy of PCIM with any existing groups. Therefore, the number of groups (PCIM's duplicates) created must be minimal. □

Algorithm 4.2.3 Identifying the Distributed Processing Components (DPCs) in the module structure graph G^*.

Input: 1. A module structure graph with module duplication G^*

2. C_i for $1 \leq i \leq |N^*|$, where C_i is the set of modules which can run concurrently with module i in G^*.

Output: A set of Distributed Processing Components (DPCs)

Procedure:

Begin

1. /*Initialization */

   a. Create an initial DPC $F_0 = \{r_0\}$, where r_0 is the root of G^*

   b. Let Q be a queue and Q = ϕ

   c. enqueue (r_0, Q)

2. /* Breadth first traversal on graph G^* */

   while Q $\neq \phi$ do

   a. Let x = dequeue (Q)

   b. for each immediate successor s of node x do

      enqueue (s, Q)

      endfor

   c. Let F be a DPC such that x is an element of F

   d. Let Child = {s | s is the immediate successor of x}

3. /* Checking if s already belongs to some DPC */

   while Child $\neq \phi$

   a. Select the module s in Child which involves the largest amount of communication with module x.

   c. Let Child = Child - {s}

   d. if s is an element of some DPC then

      goto step 3.

      endif

4. /* Finding a DPC for s */

   if s is an element of C_n for any n, an element of DPC F then

   a. Find a DPC H such that s is not an element of C_n,

where {n | n is an element of H}; create a new DPC H = ϕ,
if no such H exist

b. Let H = H ∪ {s}

else

c. Let F = F ∪ {s}

endif

endwhile

endwhile

End

Step 1 creates the initial DPC F_0 for the root and a queue Q for the use of breadth first traversal on the graph. While the content of the queue Q is not empty in step 2, we will consider the first module x in the queue and put the immediate successor of x in the queue (which is in the breadth first traversal order). The set of the immediate successors of x is named Child. Steps 3 and 4 will determine each immediate successor of x to which DPC should belong. We also locate a DPC F where we would like to be able to place these modules together to reduce the communication costs involved in their invocation. We select the module which involves the largest amount of communication with its invoker. This is done in order to minimize the communication cost and is necessary to exploit concurrency. If no existing DPC can accommodate the a successor, then a new DPC will be created.

Theorem 4.2.4 The algorithm to find the Distributed Processing Components (DPCs) in the modified module structure graph G^* takes O(MAX(ne, $n^2\log n$)) steps.

Proof of Theorem 4.2.4 Step 1 takes only constant time to create initial DPC and queue Q. Step 2 traverses the graph in the breadth first order. Because the graph

is hierarchical, the total steps for the operation enqueue are equal $O(e)$, so that the while loop at step 2 will iterate e times. Because the maximum number of DPC is the number of the nodes in the graph, step 3 at most takes n steps with e iterations, so that step 3 takes $O(ne)$ steps.

Step 4 requires that we examine each existing DPC, and for each of these we may need to examine each set C_i, where i is an element of the the given DPC. Because no node is an element of more than one DPC, in the worst case, the amounts to examine each node s in the set C_s for the node under consideration takes $O(n)$ steps. Each search can be done in $O(\log n)$ time. Thus, step 5 can be done in $O(n\log n)$ time. Note that step 5 is executed once for each node; thus, step 4 contributes a complexity of $O(n^2\log n)$ to the algorithm's complexity. □

4.2.1 Example

We will use the module structure graph shown in Fig. 4.3 as an example to demonstrate the partitioning with module duplication algorithm. Algorithm 4.2.1 will first determine all the sets of modules which can run concurrently with each module in the module structure graph. The following are each module's concurrency set.

$C_1 = \{2, 7, 8, 9, 10\}$

$C_2 = \{1, 3, 4, 5, 6, 7, 8, 9, 10\}$

$C_3 = \{2, 7, 8, 9, 10\}$

$C_4 = \{2, 7, 8, 9, 10\}$

$C_5 = \{2, 6, 7, 8, 9, 10\}$

$C_6 = \{2, 5, 7, 8, 9, 10\}$

$C_7 = \{1, 3, 4, 5, 6, 7, 8, 9, 10\}$

$C_8 = \{2, 7, 8, 9, 10\}$

$C_9 = \{2, 7, 8, 9, 10\}$

$C_{10} = \{2, 7, 8, 9, 10\}$

A module will potentially concurrently run with itself if it is a PCIM. Therefore, we can determine all the PCIMs by checking the concurrency relations among modules. We will determine all the PCIM modules using algorithm 4.2.1 as follows:

$PCIM_{set} = \{7, 8, 9, 10\}$

Algorithm 4.2.2 duplicates all the PCIM modules in the module structure graph and determines the relationships among the all the modules in the newly created module structure graph. The concurrency set for each module in the duplicated graph will also be determined in this algorithm. Fig. 4.5 is the module structure graph after module duplication. The concurrency set of each module in the new module structure graph is as follows:

$C_1 = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_2 = \{1, 3, 4, 5, 6, 7, 8, 9, 10\}$

$C_3 = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_4 = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_5 = \{2, 6, 7, 7^*, 8^*, 9^*, 10^*\}$

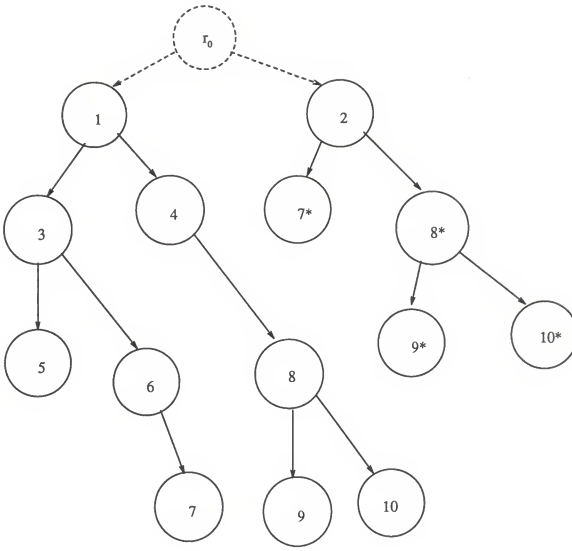
$C_6 = \{2, 5, 7^*, 8^*, 9^*, 10^*\}$

$C_7 = \{2, 5, 7^*, 8^*, 9^*, 10^*\}$

$C_{7^*} = \{1, 3, 4, 5, 6, 7, 8, 8^*, 9, 9^*, 10, 10^*\}$

$C_8 = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_{8^*} = \{1, 3, 4, 5, 6, 7, 7^*, 8, 9, 10\}$



Module Relationship:

1. ONE-OF(3,4)
2. CON(7*,8*)
3. CON(5,6)
8. SEQ(9,10)
8*. SEQ(9*,10*)

Figure 4.5. Module Structure Graph After PCIM Duplications

$C_9 = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_9{}^* = \{1, 3, 4, 5, 6, 7, 7^*, 8, 9, 10\}$

$C_{10} = \{2, 7^*, 8^*, 9^*, 10^*\}$

$C_{10}{}^* = \{1, 3, 4, 5, 6, 7, 7^*, 8, 9, 10\}$

Algorithm 4.2.3 partitions the duplicated module structure graph into distributed processing components. There will be four DPCs for the example module structure graph shown as below. Fig 4.6 is the module structure graph after partitioning.

DPC $F_0 = \{r_0, 1, 3, 4, 5, 8, 9, 10\}$

DPC $F_1 = \{2, 7^*\}$

DPC $F_2 = \{8^*, 9^*, 10^*\}$

DPC $F_3 = \{6, 7\}$

4.3 System Performance Estimation

After having derived a set of DPCs for execution at various distributed sites, we need to estimate the performance of the system. We will give the following definitions in order to estimate the system performance.

- E_i is the estimated execution time of the code of module i.
- C_{ij} is the estimated communication time between nodes i and j in module structure graph.
- SONS(i, j) is the worst case estimated execution time of all the children of j, where (i, j) is an edge in the module structure graph.

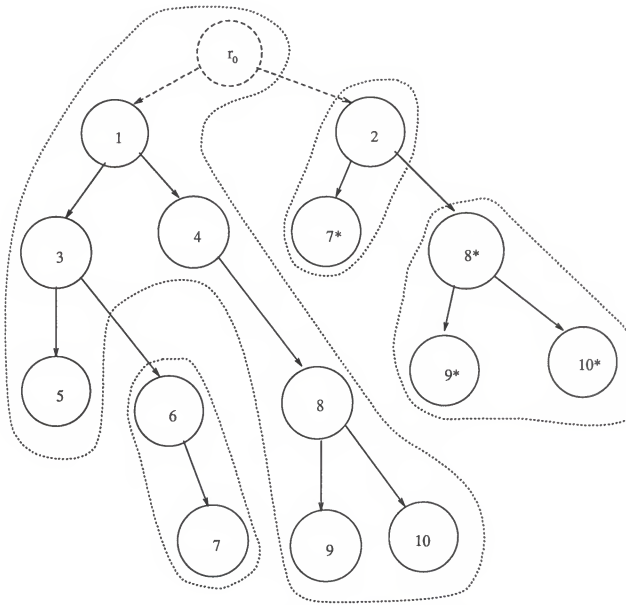


Figure 4.6. Module Structure Graph After Partitioning (With Module Duplication)

- $SUBOR_j$ is the concurrency constraints of all the subordinates of module j. It is of the form of STRING. STRING has the form of a node or $OP(m_1, m_2, ..., m_s)$ where OP is SEQ, CON or ONE-OF and each of the $m_1, m_2, ..., m_s$ may be a node or a STRING.

- F(i, j) is the sum of C_{ij}, E_j and SONS(i, j).

Algorithm 4.3.1 System performance estimation for module structure graph G

Input: Module structure graph G = (N, E, R)

DPCs derived from the design stage partitioning algorithms

E_i, F(i, j), C_{ij}

Output:The estimated system performance of G

Procedure:

Begin

1. Add a dummy node D and dummy edge that link the node D with the root node of G.
2. The estimated system performance of G = F(D, *root*)

function F(i, j)

Begin

if j is a terminal node then

F(i, j) = $E_j + C_{ij}$

else

F(i, j) = $E_j + C_{ij}$ + SONS(j, $SUBOR_j$)

endif

End

function SONS(*j:NODE; m:STRING*)

Begin

if m is of the form of a node then

SONS = F(j, m)

elseif *OP* of m is SEQ then

SONS = $\Sigma_{i=1}^{s} SONS(j, m_i)$

elseif *OP* of m is CON or ONE-OF then

SONS = maximum(SONS(j, m_i | 1 ≤ i ≤ s))

endif

End

The algorithm estimates the worst case performance of the partitioned module structure graph. Function F(i, j) calculates the execution time of module j and its descendants invoked by module i. It should be noted that if module i and j in the same DPC then the communication time $C_{ij} = 0$. Function SONS estimates the longest path of execution time invoked by a module. The execution time of a module at this stage is a rough estimation made by the designer. However, given the system performance estimation, the designer can then decide whether or not the designed system fits the design criteria, for example the system real time constraint. If the designed system is not satisfied, then the system must be redesigned.

4.4 Design Stage Allocation

After the partitioning on the module structure graph, we will get a group of distributed processing components (DPCs) which are the basic units for allocating to distributed sites. At the partitioning phase, we have already considered maximizing the potential concurrency and avoiding unnecessary message traffic among modules.

However, it is still a logical design and does not take the physical hardware configuration such as the number of distributed sites into consideration. It is the responsibility of design stage allocation to assign those distributed processing components (DPCs) to the distributed sites. The relations among DPCs, which is the information needed for the allocation strategies, can be derived from the original module structure graph by the following algorithm.

Definition 4.4.1 A DPC graph $G_d = (N, E)$ is a directed graph with set of nodes N, the set of edges E, such that

1. Each DPC is represented by one node n_i in N
2. The directed edge $(n_i, n_j) \in E$ if there is an message sent from DPC n_i to n_j

Algorithm 4.4.1 Generating DPC graph G_d = (N, E)

Input: 1. Module structure graph G = (N, E, R)
(If using partitioning without module duplication algorithm)
2. Modified module structure graph G^* = (N, E, R)
(If using partitioning with module duplication algorithm)
3. A set of Distributed Processing Components (DPCs)

Output: A DPC graph G_d = (N, E)

Procedure:

Begin

1. /* Initialization */
Let Q is a queue and Q = ϕ
enqueue (r_0, Q)
2. /* Breadth first traversal */
while Q $\neq \phi$ do

a. Let x = dequeue (Q)

b. if x is marked "visited" then

goto step 2.

endif

c. for each immediate successor s of x, do

enqeue(s, Q)

endfor

3. /* Determining the relation between two different DPCs */

for each immediate successor s of x do

a. if $s \in DPC_s$ and $DPC_x \neq DPC_s$

then

b. if edge (DPC_x, DPC_s) exists

then

c. Add the amount of communications from x to s

to the directed edge (DPC_x, DPC_s)

else

d. Create a directed edge (DPC_x, DPC_s) and mark the

edge with the amount of communication sent from x to s

endif

endif

endfor

End

The algorithm follows the breadth first order to check each module in the the module structure graph (or modified module structure graph). If module x and its

immediate successor s are in different DPCs, then add the communication volume between them to the edge (DPC_x, DPC_s) in the DPC graph. The complexity of the algorithm is $O(e)$ due to the breadth first order traversal.

Theorem 4.4.1 The DPC graph $G_d = (N, E)$ does not always have precedence relations among the DPCs.

Proof of Theorem 4.4.1 Assuming DPCs always have the precedence relations among them, then we can find counter examples. M_1, M_2, and M_3 are modules in the module structure graph. M_1 invokes M_2, and M_2 invokes M_3, however, the partitioning algorithm may cluster M_1 and M_3; in DPC_1 and M_2 in DPC_2, and then there will be no precedence relation between DPC_1 and DPC_2. It is a contradiction. □

A distributed processing component (DPC) is the basic unit for us to allocate them to the distributed sites. The allocation strategies can be categorized as static vs. dynamic, or task with precedence relations vs. without precedence relations. The types of cost function for allocation have (1) total interprocessor communication cost, (2) total execution and interprocessor communication cost, (3) completion time, and (4) load balancing. In the selection of our design stage allocation strategy for the distributed parallel computing systems, we have to consider the factors mentioned above. The design stage allocation should be a kind of static allocation because it is done before coding, and it is not possible to do it at the run time. There does not always have to be precedence relationships among DPCs which have been proven before. At the design stage, the communication volume between modules can be estimated from the message size and invocation frequency by the designers; however, the execution time of the module is not easy to estimate. That will limit the choice of cost function for the allocation strategy. Based on the constraints of

information available at the design stage, we will adopt the existing task allocation methods which are static, without task precedence relations. If the designer can make better estimation on the execution time of each DPC then he can adopt the allocation methods with the cost function of minimizing total execution and inter processor communication cost, e.g., [Lo84], or minimizing total execution time, e.g., [Shen85], otherwise he has to adopt the allocation methods with the cost function of minimizing total communication cost, e.g., [Chu80].

CHAPTER 5
CODING STAGE PARTITIONING

5.1 Overview

Coding stage partitioning partitions the program code into sequential units for parallel computation. Partitioning is necessary to ensure that the granularity of the parallel program is coarse enough for the target parallel machine without losing much parallelism. Allocation is necessary to achieve good processor utilization and to optimize interprocessor communication in the target parallel machine.

The granularity of a parallel program is the average size of a sequential unit of computation in the program with no interprocessor synchronization or communication. For a given parallel machine, there is a minimum program granularity which depends on the machine characteristics. If the program granularity is below the minimum granularity, the performance degrades significantly. It is desirable for a parallel machine to have a small granularity so that it can efficiently support a wide range of programs. It is also desirable for a parallel program to have a large granularity so that it can execute efficiently on a wide rage of parallel machines.

Besides partitioning and allocation, identifying parallelism in the program is an important problem to be solved when compiling a program for parallel execution on a multiprocessor. In general, the parallelism in the program depends on the parallel algorithm designed by the programmer, so that the problem of identifying parallelism belongs to the domain of programming languages.

It is preferable for a parallel language to be referential transparent (without side effects). The freedom from side effects makes it easier to identify parallelism. The

executability of an instruction is decided by the availability of its operands. There will be no need to do dependence analysis to identify parallelism as in the parallelization of imperative language programs. Most current program partitioning approaches are done on some kind of side-effect-free parallel languages, such as SISAL [Gaudiot89, Sarkar88] and ALFL [Hudak85]. In our approach, we will use PROOF [Yau91] as the target language. However, the applicability of our approach is not limited to the use of PROOF language; any side-effect-free language which can be translated to the computation graph is applicable for our coding stage partitioning approach.

There are three possibilities for program partitioning and allocation: (1) run time partitioning and run time allocation, (2) compile time partitioning and run time allocation, and (3) compile time partitioning and compile time allocation. In the first approach, both partitioning and allocation are postponed until run time. The advantage is the availability of run time information which may lead to a better partition and schedule. The major disadvantage in doing everything at run time is the extra overhead incurred during program execution. The second approach of compile time partitioning and run time allocation is the most common model of program execution in current multiprocessor systems. The problem is how to determine a partition at compile time which yields the smallest parallel execution time at run time, for a given parallel program and target parallel machine. In the third approach, both partitioning and allocation are performed automatically at compile time. Compile time allocation is attractive because it eliminates scheduling overhead entirely at run time. The disadvantage is that compile time estimation for execution times and overhead may be inaccurate for some program inputs, leading to inefficient schedules.

We will use compile time partitioning and run time allocation in our approach. The process of coding stage partitioning and allocation is illustrated in Fig. 5.1. To generate the target code for a specific parallel machine, a PROOF program is first

translated into an intermediate form which is a graphical representation of the program and independent of the underlying multiprocessor architecture. The graphical representation preserves all the potential parallelism in the program and can represent the data dependency in the program. This computation graph is first used to generate profile information using the profile generator. The profile information will contain (1) the average number of calls to each function, (2) the average execution frequencies for nodes in the graph, and (3) the average data size communicated between each node. With the specific information about the underlying multiprocessor system, such as communication overhead and number of processors, the computation graph is analyzed by our coding stage partitioning algorithm and is divided into partitions which have the most suitable granularity for the underlying parallel machine. The partitioned computation graph then is used to generate executable tasks. We will use run time allocation in our approach. At the run time, the executable tasks are scheduled to the processors according to the allocation strategy.

5.2 The Graphical Representation

In most compilers, the front end of the compiler translates a source program into an intermediate representation from which the back end of the compiler generates target codes. Details of the target language are confined to the back end, as far as possible. The benefits of using a machine-independent intermediate form are

1. Retargeting is facilitated: a compiler for different machines can be created by attaching a back end for the new machine to the existing front end.

2. A machine-independent code optimizer can be applied to the intermediate representation.

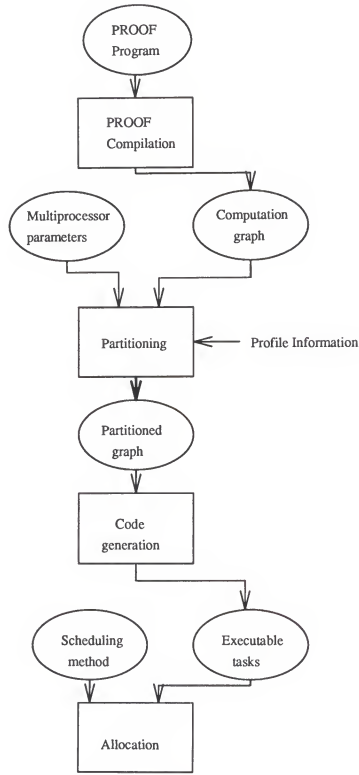


Figure 5.1. Coding Stage Partitioning and Allocation Using PROOF Programs.

The intermediate form usually can be represented graphically. In our approach, coding stage partitioning will be done by the analyzing of the intermediate form. We use the graphical representation of the intermediate form and categorize the instructions of intermediate codes into four types. We will use PROOF as an example; however, any parallel language which has the referential transparency property and can be translated to the graph representation discussed in this section can use our coding stage partitioning algorithm. In our approach, PROOF parallel programs are represented by graphical representation as an intermediate form in the compiling process. Execution profile information is annotated to the intermediate form for the use in the partitioning process. A PROOF program is a set of objects PROG = $\{O_1,O_2,...,O_n\}$, where n is the number of objects in the program. An object consists of a set of methods. Each method is a function which is represented by a pair (g,f), whereas g is a computation graph (CG) representing the computation of the method and f is a frequency count which gives the average number of calls to this method in a single execution of the program PROG. The frequency count f comes from the the profiling information which we will discuss in the next section.

Definition 5.2.1 An object consists of a set of methods, Object = $\{(g_1,f_1), (g_2,f_2), ..., (g_m,f_m)\}$, where m is the number of methods in the object. A method (g_i,f_i) consists of

1. a CG graph g_i representing method i's computation, and
2. f_i, a frequency count which gives the average number of calls to method i in a single execution of program PROG.

Definition 5.2.2 A CG graph is a 4-tuple G=(N, E_C, F_C, F_T), where

1. N is a set of nodes. CG graphs can have the following kinds of nodes:

(a) A *simple* node $n_s \in$ N represents an indivisible sequential computation.

(b) A *function call* node $n_u \in$ N contains a function name and represents a call to that CG function.

(c) A *compound* node $n_c \in$ N contains a set of graph-frequency pairs, $n_c.s = \{(n_c.g_1, n_c.f_1),(n_c.g_2, n_c.f_2), ...,(n_c.g_k, n_c.f_k)\}$, where $n_c.g_i$ is a CG graph and $n_c.f_i$ is $n_c.g_i$'s average execution frequency in n_c. Each execution of n_c is an arbitrary sequential execution of its subgraphs, so that subgraphs may be executed any number of times in any order. If we consider the subgraph to be like basic blocks, the compound nodes correspond to the instructions like If, While.

(d) INPUT and OUTPUT are two distinguished nodes which identify the import and export edges of the graph.

2. E_C is a set of communication edges. An edge $e \in E_C$ is the pair $((n_a,p_a),(n_b,p_b))$ where

(a) $n_a \in$ N is the node from which the edge, originates.

(b) p_a is the output port of n_a from which the edge originates,

(c) $n_b \in$ N is the node at which the edge terminates, and

(d) p_b is the input port of n_b at which the edge terminates.

The communication is assumed to occur after node n_a completes execution and before node n_b starts. Therefore, communication edges also enforce precedence constraints among nodes.

3. F_C is the communication cost function. $F_C(n_a,p_a)$ gives the average data size produced on output port p_a of node n_a, and transferred along any communication edge $((n_a, p_a), (n_b, p_b)) \in E_C$, in a single execution of CG graph G. Data size is measured in some predetermined unit like bits, bytes, or words. A communication edge may have no data if used solely to enforce a precedence constraint. The data size of such an edge is assumed to be zero.

4. F_T is the execution time cost function. $F_T(n_a)$ gives the average sequential execution time of node n_a, in some predetermined unit like machine cycles or nanoseconds.

A CG graph contains information about program structure, parallelism, execution frequencies and costs for communication, and execution time. The structure of the CG graph for each function can be derived from the compiling process. The CG graph also contains information regarding the precedence relationship between nodes. Because the PROOF language is referential transparent and without side effects, the execution of the node depends on the availability of the data importing from the precedent nodes. The execution frequencies for functions and subgraphs of compound nodes are usually unavailable from the program text. Our approach is to generate these values from an execution profile of the program. The communication cost function F_C, which given average data size communicating between nodes, also will come from the profile information of the program. With the information about the specific parallel machine and the communication cost function F_C, the communication time for the average data size between nodes can be derived. The execution time cost function F_T is a machine dependent parameter in the CG graph. With a specific parallel machine, the execution time of the simple nodes is determined by the machine primitive instructions. Execution time of compound nodes and function nodes will

be derived from the execution time of simple nodes and the profile information of execution frequency.

Fig. 5.2 is an illustration of a CG graph. The CG graph represents the computation of the merge sort of elements in an array. The algorithm has recursive function call on itself. This CG graph contains all the types of nodes - INPUT and OUTPUT, simple, compound, and function nodes. The node which computes array size is a simple node. The node to compare if an array with only one element contains two subgraphs is a compound node. The nodes of split_array, merge_array, and merge_sort are function nodes. With the annotation of the profile information and the specific information on the target parallel machine, we will be able to do the partition on the CG graph.

5.3 Execution Profile Information

Execution profile information plays an important role in building a CG graph for the PROOF programs. It provides average data sizes for all communication edges and average frequency values for function call and subgraphs of compound nodes. For efficiency and convenience, execution time costs are not generated directly in the profile, but are computed by using frequency values. Unlike execution time, the profile information needed consists only of counts and sizes which can be measured exactly. Also, this information can be generated by any (possibly sequential) execution of the program on any machine (possibly different from the target).

Execution profiles have traditionally been used to help a programmer identify the most frequently executed and time-consuming parts of a program [Wall91]. Optimization of these statements yields the greatest improvement in the program's execution time. An important issue in the use of execution profiles is the sensitivity to changes

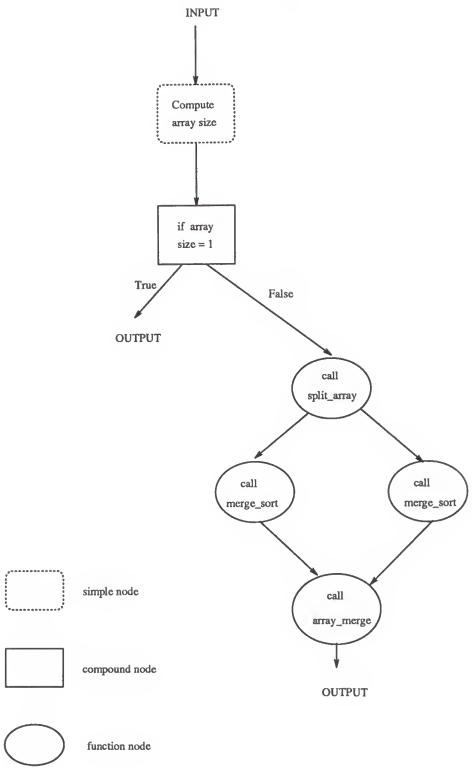


Figure 5.2. Computation Graph for Merge Sort

in profile information due to different program inputs. A large sensitivity is undesirable because the profile tailors to the program inputs used to generate frequency counts and data sizes to estimate execution time and communication costs. Most optimization uses just profile information to identify the most frequently executed instructions. For many programs, this use is reasonably insensitive to program inputs. We use profile-generated frequency counts and data sizes to estimate execution time and communication costs.

The profile information needed for partitioning consists of

1. average number of calls to each function in the program,
2. average execution frequencies for all CG subgraphs in compound nodes, and
3. average data sizes at all output ports of all nodes.

The average number of invocations to each function can be measured by executing special codes to increment counters at appropriate positions in the program.

Definition 5.3.1 Let F(g) be the total frequency count of CG graph g, over a single execution of the program. Then for any compound node n_c in CG graph G, the average frequency of its i^{th} subgraph is given by

1. $n_c.f_i = 0$, if $F(G) = 0$,
2. $n_c.f_i = F(n_c.g_i)/F(G)$, otherwise.

The average frequency of a subgraph in a compound node is the probability to execute that subgraph in the execution of the compound node. Average data sizes also can be accumulated by using counters. A special code is required to evaluate the size of a datum at run time.

Definition 5.3.2 Let $S(n_a, p_a)$ be the size of all data produced at output port p_a on node n_a, over a single execution of the program. Then the average data size for a single execution of G is given by

1. $F_C(n_a,p_a) = 0$, if $F(G) = 0$,
2. $F_C(n_a,p_a) = S(n_a,p_a)/F(G)$, otherwise.

Execution profile information can be improved by averaging over several program inputs. The main reasons for not generating F_T in an execution profile are

1. Measurement of execution times in an execution profile is usually an approximation, unlike frequency counts which are exact.
2. Further timing inaccuracies occur when the execution time of the profiling code itself contributes to the execution times being measured.
3. Execution time measurements are valid only for the machine on which the profile which was generated. Frequency counts do not have this restriction. By computing rather than measuring average execution times, the same frequency profile information can be used to derive average execution times for different target architectures.

There are many advantages for using frequency count profile instead of execution time profile as mentioned above. However, we have to develop methods for deriving the execution time from the frequency count. The average times of simple nodes are the starting values from which all other F_T values are derived. A simple approach to calculate simple node execution time is to just add the execution times of the target instructions which implement the simple node.

The average execution time of a compound node is determined by the following rule:

$$F_T(n_c) = \Sigma_i n_c.f_i \text{ x } F_T(n_c.g_i)$$

which is the sum of the products of each subgraph's average frequency and execution time of that subgraph.

The average execution time of a function node (CG graph g) is determined as:

$$F_T(\mathrm{g}) = \Sigma_{n \in N} F_T(\mathrm{n})$$

which is the sum of all its nodes in the CG graph representing that function.

The average time for function calls gets complicated for the recursive function calls. However, if we profile the average depth of recursive calls and amortize the the cost to each recursive call, we can derive the execution time of a function node with recursive calls. Before giving the estimation formula for the recursive calls, we will give the following definitions:

1. B(j)= base cost of function call node j, assuming all internal calls in j have zero cost (assuming all the cost of recursive function calls in node j is zero).

2. E(j)= approximation for the cost of an external call to function j (with the cost of recursive functions).

3. I(j)= approximation for the cost of an internal call to function j (cost of all the recursive functions in the node j).

4. t_j= the total number of calls to function j in the entire program.

5. i_j= the total number of internal calls to function j (the depth of the recursive call).

6. e_j= the total number of external calls to function j is given by $e_j = t_j - i_j$. (the number of invocation of node j by other nodes).

The profile information already includes the execution frequency of each function. Of the value defined above, t_j, i_j can be derived from the profile information by modifying the profiler to distinguish the source of invocation if the function is itself. The base cost of function call B(j) can be derived from the CG graph of g by using the formula mentioned above for execution time estimation of function nodes without considering recursive call. The total execution time for calling function j is e_jE(j); however, if we in line expand all the recursive call in function j, the total execution time of the expanded CG graph should be the same as for the previous one. We can derive the relationship between E(j) and B(j) as follows:

e_jE(j) = t_jB(j)

From the formula above, we can get the approximation for the cost of each external call to function j. If we amortize the cost of recursive calls and assume that all internal calls have the same cost, then the difference of e_j x (E(j) – B(j)) will be the same as the total internal call execution time i_j x I(j). The formula is shown as follows:

E(j) – B(j) = (i_j/e_j)I(j)

where B(j) can be derived from the CG graph of g, E(j) can be derived as the previous formula, and i_j and e_j come from the value of profile information on CG graph g, so that the estimation of computation cost of each recursive function call can be derived from the above formulas.

Our profile information consists of (1) average number of calls to each function, (2) average execution frequencies for subgraphs in compound nodes, and (3) average

data size at output ports in each node. They are independent of the target parallel machine. When we transform this machine-independent information into the execution time and communication overhead on a specific machine, we will need some characteristic parameters of that machine. We will model a parallel machine as a 5-tuple M = (P, R_C,W_C,F_S,T_{sched}), where

1. P is the number of processors in the parallel machine.

2. R_C is the communication overhead function for reading data. R_C(i, j, s) is the execution time incurred by processor j to receive s units of data from processor i, j≠i.

3. W_C is the communication overhead function for writing data. W_C(i, j, s) is the execution time incurred by processor i to send s units of data to processor j, i≠j.

4. F_S is the simple node execution time cost function. $F_S(n_s)$ gives the execution time of simple node n_s. F_S is used by the cost assignment algorithm to provide the F_T values of all nodes.

5. T_{sched} is the scheduling overhead (in execution time units) incurred by a processor when it begins execution of a new task.

In the real MIMD parallel machines, there also will be communication delay overhead in the interprocessor communication; however, we will use the list scheduling strategy for the processor allocation. List scheduling assigns tasks whose predecessors have been completed to the next available processor; therefore, our run time scheduling model does not offer any opportunity for overlapping communication delay with computation in a processor, since the next task to be executed on a processor

is determined only after the processor completes executing its current task. We do not use the delay overhead, but instead assume that all the communication delay is included in the reading and writing overhead functions R_C and W_C.

5.4 Objective Function of Partitioning

Before describing our partitioning algorithm, we will give the definition of cost function for partitioning a CG graph.

Definition 5.4.1 For a CG graph g, with a task partition Π and a target parallel machine M, define

1. $T_{seq} = F_T(g)$, the average sequential execution time of CG graph g. T_{seq} does not contain any scheduling or communication overhead.

2. $T(\tau_i) = \Sigma_{n \in \tau_i} F_T(n)$, the average sequential execution time of task $\tau_i \in \Pi$. Like T_{seq}, $T(\tau_i)$ does not contain any overhead.

3. $Q(\tau_i)$ = the average frequency of task τ_i in CG graph g.

4. $E_{IN}(\tau_i)$ = the set of data needed by τ_i.

5. $E_{OUT}(\tau_i)$ = the set of data produced by τ_i.

6. $T_{in}(\tau_i) = \Sigma_{(n_a,p_a) \in E_{IN}(\tau_i)} R_C(F_C(n_a,p_a))$ is the average input communication overhead for task τ_i.

7. $T_{out}(\tau_i) = \Sigma_{(n_a,p_a) \in E_{OUT}(\tau_i)} W_C(F_C(n_a,p_a))$ is the average output communication overhead for task τ_i.

8. $O(\tau_i) = T_{sched} + T_{in}(\tau_i) + T_{out}(\tau_i)$ is the total average overhead for task τ_i.

9. $T_{total}(\Pi) = \Sigma_{\tau_i \in \Pi} Q(\tau_i) \times (T(\tau_i) + O(\tau_i)) = T_{seq} + \Sigma_{\tau_i \in \Pi} Q(\tau_i) \times O(\tau_i)$, is the average total execution time of all tasks in Π.

10. $T_{crit}(\Pi)$ is the estimate of the critical path length of the tasks at run time, using $T(\tau_i)+O(\tau_i)$ as the estimate of task τ_i's execution time (including overhead)

To find an optimal partition on computation graph g with minimal execution time is a NP problem, and if using the scheduling methods with no unforced idleness to allocate the node in the graph, the parallel execution time T_{par} of the program on a P processors parallel machine is bounded as

$MAX(T_{crit}, T_{total}/P) \leq T_{par} \leq 2 \times MAX(T_{crit}, T_{total}/P)$

This have been proven by Sarkar [Sarkar87]. It is intractable to find the execution time partition, so that we will use the $MAX(T_{crit}(\Pi), T_{total}(\Pi)/P)$ as an estimation to find the partition in the CG graph. The objective is a partition of the graph to minimize the objective function.

Definition 5.4.2 $F(\Pi)= MAX(T_{crit}(\Pi), T_{total}(\Pi)/P)/(T_{seq}/P) = MAX(T_{crit}(\Pi) \times P/T_{seq}, 1+(\Sigma_{\tau_i \in \Pi} Q(\tau_i) \times O(\tau_i)/T_{seq}))$ is the cost of partition Π.

The cost function is defined to be the maximum of two terms:

1. The critical path term, $T_{crit}(\Pi)/(T_{seq}/P)$, which is the estimated critical path of the partitioned program, normalized to T_{seq}/P, the ideal parallel execution time on P processors.

2. The overhead term, $1+\Sigma_{\tau_i \in \Pi} Q(\tau_i) \times O(\tau_i)/T_{seq}$, which equals 1 plus the estimated total overhead in the program, normalized to T_{seq}/P.

We can see that objective function $F(\Pi)$ expresses the trade-off between parallelism and overhead. If the partition is too fine , the overhead term $\Sigma_{\tau_i \in \Pi} Q(\tau_i) \times O(\tau_i)$ will be large, causing $F(\Pi)$ to be large. If the partition is too coarse, then $T_{crit}(\Pi)$ will be large due to the loss of parallelism, causing $F(\Pi)$ to be large again. $F(\Pi)$ is minimized at an optimal intermediate granularity.

5.5 Partitioning Algorithm

The partition cost function F(Π) is the MAX of two terms:

1. the critical path term, $T_{crit}(\Pi)$xP/T_{seq}, and
2. the overhead term, $1+\Sigma_{\tau_i \in \Pi}\ Q(\tau_i)xO(\tau_i)/T_{seq}$.

The overhead term is monotonically nonincreasing with i, since a move to a coarser granularity cannot increase the total overhead, $\Sigma_{\tau_i \in \Pi}\ Q(\tau_i)xO(\tau_i)$. The critical path term is more erratic. Each critical task, τ_c, contributes $T(\tau_c)+O(\tau_c)$ to $T_{crit}(\Pi)$. As i increases, the $O(\tau_c)$ terms in critical tasks tend to decrease due to reduced overhead, but the $T(\tau_c)$ terms tend to increase due to sequentialization. However, the value of cost function will be minimum at some point. The partition at that point is what we are seeking.

Our partitioning algorithm is to start with an initial fine granularity partition, Π_0, and then iteratively merge tasks until we reach the point that the trade-off between parallelism and communication overhead is optimal. Beyond that point, any further merge of tasks will lose more parallelism than the gain from reducing communication overhead.

The following is the algorithm for partitioning.

Algorithm 5.5.1 Finding the partition on the computation graph G with minimum cost function value.

Inputs:

1. A CG graph, G=(N, E_C, F_C, F_T)
2. An initial fine granularity partition, Π_0

Outputs:

1. Partition Π_{min}(G) with the smallest F(Π)

Procedure:

Begin

1. Let $\Pi_{current} = \Pi_0$, $\Pi_{min} = \Pi_0$

   Let Overheadsum = $\Sigma_{\tau_i \in \Pi_0}$ Q(τ_i)xO(τ_i)

2. While $|\Pi_{current}| > 1$ do

   a. Calculate the critical path CP in $\Pi_{current}$

   b. Find τ_a, $\ni$ (Q(τ_a)xO(τ_a)) $\geq$ (Q(τ_i)xO(τ_i)), $\forall$ $\tau_i \in$ CP

   c. Find τ_b such that edge between τ_a and τ_b

      has the largest communication overhead.

   d. Merge task τ_a and τ_b into a new task τ_{new}.

      and create new partition Π_{new}, $\Pi_{current} = \Pi_{new}$

   e. Overheadsum = Overheadsum -

      (Q(τ_a)xO(τ_a)) - (Q(τ_i)xO(τ_i))+ (Q(τ_{new})xO(τ_{new}))

   f. Let F(Π_{new}) = MAX{CP(τ_{new})xP/$F_T(\tau_{new})$,

      1 + Overheadsum/$F_T(\tau_{new})$)

   g. if F(Π_{new}) < F(Π_{min})

   then

      Let $\Pi_{min} = \Pi_{new}$

   endif

   endwhile

3. Reconstruct partition Π_{min}

End

Step 1 in the previous algorithm initializes the $\Pi_{current}$ to the initial finest granularity and the total overhead. Step 2a finds the critical path in the computation graph. We try to reduce the largest overhead by clustering the partitions in the critical path involved with the largest communication between them. Step 2b finds the partition τ_a in the critical path which have the largest overheadsum. In step 2c, we will determine the partition τ_b which contributes the most to the overhead of τ_a. We cluster τ_a and τ_b to reduce the overhead in steps 2d. Step 2e and 2f recalculate the value of the new graph partition under the objective function. Step 2g checks if the new partition generates better objective function; if it does, we set the Π_{min} to Π_{new}. Step 3 determines the partition which has the smallest value of cost function and reconstructs the partition Π_{min}.

Theorem 5.5.1 The algorithm for finding the partition on the computation graph G with minimum cost function value requires $O(ne)$ steps where n is the number of nodes and e is the number of edges.

Proof of Theorem 5.5.1 Step 1 is for initialization. The calculation for Overheadsum takes n steps. Step 2 takes at most n iterations in the worst case that all nodes of G are clustered into the same partition. Step 2a will take $n+e$ steps to find the critical path using the available critical path algorithm in the graph theory. Step 2b is to find the node in the critical path with largest overhead that takes at most n steps. Other steps in the algorithm take constant time. Therefore, the total complexity of the algorithm is dominated by step 2a's complexity $O(ne)$. □

5.6 Coding Stage Allocation

After partitioning, each partition in the CG graph is a task for allocation. There exists precedence relationship and communication overhead between tasks. The allocation problem is to assign tasks in the partitioned program to processors, so as

to minimize the parallel execution time. We assume that the target parallel machine executes one program at a time, because without this assumption, the partitioning and allocation problem also will depend on the job mix for the system. Our partitioning objective function is based on the upper and lower bounds of the parallel execution time in a multiprocessor. The assumption is that once allocated, a task could always uninterruptedly run to completion, and the allocation methods should be no unforced idleness, which means a processor never stays idle if there is a task ready for execution. The basic steps for each processor will be

1. Getting a ready task from the scheduler. All the immediate predecessors of the ready task should have finished their execution, and all the needed input data are available.

2. Fetching the ready task's input data from the processor where its predecessors store the input data.

3. Executing the ready task.

4. Storing the ready task's output data.

The run time scheduler will be responsible for enforcing all the data dependencies and control dependencies among tasks. The data dependencies are enforced by declaring a task to be ready only when all its input data have been completed. The control dependencies are enforced by creating a task when the control conditions for its compound node become true (e.g., the IF statement before deciding whether the task is in the THEN or ELSE parts should be created). However, any control flow within a task is performed by the task's sequential code and not by the run time scheduler. The run time scheduler can be an active entity like a process, or it may

be just a passive data structure shared by all processors and updated by each processor in the steps mentioned above. The partitioning algorithm is concerned only with the scheduling overhead parameter T_{sched}, which depends on both the target parallel machine and the implementation method of the run time scheduler. Usually a distributed scheduler is most efficient for large-scale multiprocessors. However, a centralized scheduler may be more appropriate for loosely coupled systems. The idea is to use the best scheduling strategy for the given parallel machine, and then let the partitioning algorithm adapt to the scheduling overhead appropriately.

The selection of allocation method will be limited to the category of heuristics which has property mentioned above. We will adopt the existing available suitable list scheduling method for our task allocation. There are heuristics which schedule tasks with precedence relationship and no unforced idleness for the processors [Graham69, Kasahara84, Hwang89]. They have the worst case bound for the execution time of tasks. However, some do not consider the communication overhead. We will adopt the Earliest Task Fist (ETF) heuristics [Hwang89] as an example for our task allocation, because it also considers the communication overhead between tasks and still has the worst case bound.

ETF uses a centralized run time scheduling approach. In ETF, the ready task which can be scheduled to start its execution at an earliest time is given the highest priority for the current scheduling decision. This allocation method tries to keep processors busy and to shorten communication delays at the same time by selecting tasks to run on free processors as soon as possible. A task is called ready if all its predecessors have been finished at that moment; however, a ready task may not be able to start its execution immediately due to communication delays. The starting time of a ready task is determined by several factors: when its preceding tasks are

finished, how long the communication delay take, and where the task and its predecessors are allocated. A ready task, in general, could start its execution at different times on different processors. The smallest starting time of a ready task is tentatively assigned to the task and is called "tentative start time". The processor which makes the "tentative start time" possible is called "tentative allocation". The ready task which has the smallest tentative start time is selected and is called the "selected task". If more tasks become ready after the earliest ready task is selected and among newly ready tasks, one may have smaller tentative start time than that of the selected task. In that case, the decision of scheduling the selected task is postponed and the selection process is reiterated.

5.7 Performance of the Algorithm

In this section we will present some performance results of our coding stage partitioning algorithm. Due to the lack of a PROOF compiler in the distributed parallel environment, we use the simulation approach to obtain the performance results. We generate the computation graph with the pattern of problems such as merge sort, and Fast Fourier Transformation. These problems generally are used for the performance evaluation of parallel algorithms. The main purpose of this implementation is to build a prototype of a partitioner as well as a simulation system to test the effectiveness of the partitioning algorithm presented in this chapter. The following parameters are used in these simulation experiments:

1. The nodes of computation graph are set to a number around 200.

2. The number of processors, P, is varied in the range 1–10.

3. Communication overhead is assumed to be entirely in the input and output components. The communication time is assumed to be linearly proportional

to the communication data size between nodes. The communication data size between nodes is randomly generated, within the specified ranges.

4. The execution time of the node in the computation graph is randomly generated within the specified ranges. We also assume that we have expanded all the function nodes, so that the frequency associated with each node is set to 1.

The computation graph goes through the partitioning phase and execution simulation phase. In partitioning phase, the computation graph is partitioned according to our coding stage partitioning algorithm with specified processor number P and scheduling overhead T_{sched}. In the execution simulation phase, the simulator schedules the partitioned computation graph to the specified P processors in a first in first out order to simulate the parallel execution. The simulation outputs give the parallel execution time and the speed up observed.

The following figures contain execution time speed up plots for different experiments. The speed up is the ration of the parallel execution time on P processors to the program's sequential execution time. Fig. 5.3 is the execution time speed up of the computation graph with a pattern similar to merge sort. The sequential execution time is normalized in order to be compared with other experiments. We use different communication overhead parameters (Low: 1 time unit/data size, Medium: 5 time unit/data size, High: 10 time unit/data size). The speed up curves are drawn in solid line. The speed up is worsened as the communication overhead is increased. We also use the computation graph without partitioning to simulate its execution time in order to see the performance of the partitioning algorithm. We draw the speed up curves in dashed lines. The partitioning algorithm does improve the performance of the speed-up.

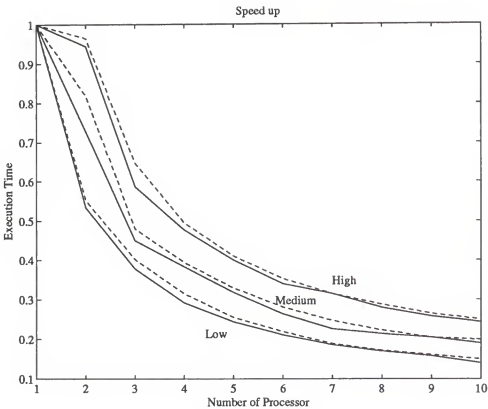


Figure 5.3. Speed Up Curves of Merge Sort

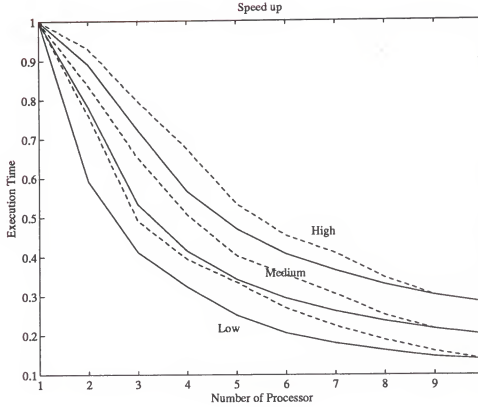


Figure 5.4. Speed Up Curves of FFT

Fig. 5.4 uses the computation graph with the pattern of FFT. The sequential execution time also is normalized to one and it uses the same parameters as the previous experiment. It also shows the same trend that when communication overhead increases, the speed up is worsened. However, at the same parameters, when the processor numbers increases (with 9 or 10 processors), the execution times are the same. However, this is most probably due to the unrealistically small input sizes of the computation graph used in the simulations.

CHAPTER 6
CONCLUSIONS

6.1 Major Results

Software partitioning and allocation are important issues in the distributed or parallel environments. Partitioning deals with establishing boundaries in the software system to exploit potential parallelism. We utilize the boundaries established in the partitioning stage to aid us in allocating software components to the processing elements. Partitioning can be done at the design stage or coding stage. Currently, most researches for partitioning are on program partitioning, while a few researches are on design stage partitioning. None of the researches uses both design stage and coding stage partitioning approaches.

In this dissertation, we integrate a two-stage partitioning strategy with the software development process for distributed parallel computing systems. The two-level parallelism property in distributed parallel computing systems, parallelism between distributed sites and parallelism within the parallel machine, is a good area to apply our two-stage partitioning approach.

We develop a design specification method for the distributed parallel software. The method is an extension of the concept of concurrently executable modules [Weber86, Weber88] using the PROOF computation model [Yau91]. The specification can be used to specify the application with the cooperate property which has more than one active control module. Different from the module in the structure chart of SADT, the module specification in our approach is an abstract data type which is more in compliance with software engineering principles.

Two design stage partitioning algorithms - - partitioning without module duplication, and partitioning with module duplication - - are developed in this dissertation. The criteria for our design stage partitioning is to exploit potential concurrency that exists among modules and to avoid nonprofitable message traffic. Shatz and Yau [Shatz86] also developed an module partitioning algorithm with the same criteria; however, our algorithm has better worst case complexity ($O(ne)$ vs. $O(n^3)$) and can apply to application with more than one control module. The algorithm of partitioning with module duplication can be used in the environments which have enough computing resources in exchange for better performance. The algorithm of partitioning with module duplication uses the same partitioning criteria as the previous one. The key point is to minimize the module duplications (without any unnecessary duplication). Yau and Wiharja [Yau91c] extend the algorithm of Shatz and Yau [Shatz86] with module duplication which is based on the structure chart of SADT. The algorithm inherits the problem of a structure chart which is single rooted and can not be used for application with more than one control module in the distributed environment. Our algorithm also has better worst case complexity than that algorithm. We can use the existing available allocation algorithms which have the property of static and without precedence relations to allocate DPCs to distributed sites.

Coding stage partitioning partitions the program code into sequential units for parallel computation. Partitioning is necessary to ensure that the granularity of the parallel program is coarse enough for the target parallel machine without losing much parallelism. We develop an algorithm for coding stage partitioning by clustering together the tasks with largest communication overhead in the critical path. The method is to partition the computation graph which is in intermediate form in the compiling process. Although we use PROOF as the target language, any other

parallel language which has the property of referential transparency and can be translated to the computation graph can use our algorithm for program partitioning. We use profile information to estimate the execution time and communication overhead. The profile information gives us more accurate information for program partitioning; however, it also limits our approach for those programs that are sensitive to the change of input data.

Our two-stage approach takes the information about potential parallelism in the design stage and coding stage into consideration. At the design stage, the potential parallelism information comes from the software designer's understanding about the problem domain. At the coding stage, the information of parallelism comes from the analysis of the program. These two kinds of parallelism information are essential for a better partitioning and should be integrated into the software development process.

6.2 Future Research Directions

This dissertation provides a starting point for future work in software partitioning for the software development of distributed parallel computing systems. Some of the possible directions for future works are as follows:

1. Our two-stage partitioning and allocation approach should be implementation a distributed parallel computing environment. The partitioning and allocation are very complex problems. We have done the complexity and performance analysis for the partitioning algorithms we developed; however, the performance of this approach needs to be tested in the real environment. Building a PROOF compiler and profiler for a target parallel machine and developing software through the whole software development process using our two-stage partitioning algorithm are needed. The software then have to be tested in the

real distributed parallel computing environment to see the effectiveness of our approach.

2. Improvements in the partitioning algorithms are necessary. Our current design stage partitioning methods will cluster the the modules in order to exploit potential concurrency and avoid unnecessary nonprofitable message traffic between modules. However, precedence relationship not always exists between the partitions. The precedence relationship is important information for making decisions in the allocation process. It is desirable to improve the partitioning algorithm to preserve the precedence relationship between partitions. The coding stage partitioning algorithm is a kind of approximation, and we can informally call one algorithm more optimal than another if it finds solutions with better values of the objective function. It would desirable to design algorithms which are more optimal than the approximation algorithms presented in this dissertation.

3. The feasibility of other better allocation methods for the proposed partitioning algorithms should be explored. We use existing allocation methods for our partitioning algorithms. It is desirable to explore allocation algorithms more suitable for our partitioning methods. For example, we use the run time approach for our coding stage allocation method in this dissertation. However, compile time scheduling has some advantages, because it entirely eliminates scheduling overhead entirely at run time. The disadvantage is that the compile time estimation of execution times and overhead may be inaccurate for some programs, leading to inefficient schedules. It is worth studying the feasibility of developing a better compile time allocation method for our coding stage partitioning algorithm.

APPENDIX
THE COMPUTATION MODEL PROOF

Our proposed approach is based on the PROOF computation model. In PROOF, a program is represented as a set of objects (active or passive) which can be executed in parallel. The major features of PROOF are (1) Class, object, and inheritance are supported in PROOF with the restriction that all the methods in objects are applicative functions. In other words, the functional paradigm is adopted at the methods definition level. (2) The guard of a method is introduced to support the synchronization between concurrent objects. Furthermore, methods along with their guards are inheritable. (3) Objects are persistent in PROOF. The following is a brief description of features in PROOF.

Applicative Functions

Methods in PROOF are purely applicative functions or functional forms, i.e., high order functions. We use a constructor $[x_1, x_2, \ldots, x_n]$ to denote a sequence of homogeneous or heterogeneous elements. In the case of homogeneous elements, it denotes a *list* or an *array* whose types are T^* and $T^n (\equiv \underbrace{T \times T \times \ldots \times T}_{n})$, respectively. In the case of heterogeneous elements, it denotes a *cartesian product* whose type is $\prod_{i=1}^{n} T_i$ $(\equiv T_1 \times T_2 \times \ldots \times T_n)$.

PROOF assumes that there is a set of primary functions and functional forms from which other functions and functional forms can be easily constructed. The following are some of the functions and functional forms.

a) Functional form α (called *apply to all*)

Type: $(T_1 \rightarrow T_2) \rightarrow T_1^* \rightarrow T_2^*$

$$\alpha f[x_1, x_2, \ldots, x_n]$$
$$\equiv [f(x_1), f(x_2), \ldots, f(x_n)]$$

α has two parameters, a function of type $T_1 \rightarrow T_2$, and a list of homogeneous elements of type T_1. The function f is applied to each element in the list and yields a list of elements of type T_2.

b) Functional form β (called *distributed apply*)

Type: $\prod_{i=1}^{n}(T_i^{(1)} \rightarrow T_i^{(2)}) \rightarrow \prod_{i=1}^{n} T_i^{(1)} \rightarrow \prod_{i=1}^{n} T_i^{(2)}$

$$\beta[f_1, f_2, \ldots, f_n][x_1, x_2, \ldots, x_n]$$
$$\equiv [f_1(x_1), f_2(x_2), \ldots, f_n(x_n)]$$

β has two parameters, a list of functions in which each function f_i is of type $T_i^{(1)} \rightarrow T_i^{(2)}$, and a list of heterogeneous elements in which the ith element is of type $T_i^{(1)}$. Each function in the first list is applied to the corresponding element in the second list. It yields a list in which the ith element is of type $T_i^{(2)}$.

c) Function γ (called *filter*)

Type: $\mathbf{bool}^n \rightarrow T^n \rightarrow T^k, 0 \leq k \leq n$

$$\gamma[b_1, b_2, \ldots, b_n][x_1, x_2, \ldots, x_n]$$
$$= \begin{cases} [\,] & \text{if } n = 1 \text{ and } b_1 = \texttt{False} \\ [x_1] & \text{if } n = 1 \text{ and } b_1 = \texttt{True} \\ \gamma[b_1 \ldots b_k][x_1 \ldots x_k] \circ \gamma[b_{k+1} \ldots b_n][x_{k+1} \ldots x_n] & \text{if } n > 1 \end{cases}$$

Here, $\circ$ denotes the concatenation of two lists. It is written in infix form for the sake of readability. γ has two parameters, a list of booleans and a list of any elements. This function yields a subsequence of the second list by selecting elements whose corresponding elements in the first list are `True`.

Because all the functions in PROOF are applicative, fine grain parallelism is obtained from the following two sources:

P1. Parallel evaluation of arguments of functions: It is made possible because of the *referential transparency* of applicative functions.

P2. Parallel evaluation of a number of functions or replications of a function: It is made possible by functional forms such as α and β.

Classes and Objects

Class Interface and Definition

In PROOF, every object is an instance of a class. A class is a template for a set of objects bearing similar behavior, and it is defined as *a generic abstract data type.* A class is defined by its interface and definition. The class interface describes the types, or signatures, of the methods provided by the class. The class definition consists of the composition of its local data and the definition of each of its methods, which are purely applicative functions.

It is important to point out that restricting the methods of classes to be applicative functions does not restrict the expressive power of PROOF, but merely requires that all the effects of the methods be explicitly specified via the parameters of the methods.

Inheritance and Genericness

Both inheritance and genericness are supported in PROOF. Inheritance is used to define a subclass as a specialization of a superclass. In a subclass, all the local data and the methods of its superclass are inherited. With additional local data, new methods may be introduced. The inherited methods also may be overridden by a new definition of the method. Genericness is used to define parametric or generic classes. An instantiation of a generic class is obtained by assigning values to the parameters of the generic class. All the local data and the methods of the generic class are inherited.

Active and passive objects

A program in PROOF consists of a set of objects. Each object is an instance of a class, and can be either *passive* or *active*. A passive object acts like a service agency. It waits passively until one of its methods is invoked by some other objects. A passive object may in turn invoke methods in other objects. An active object is active initially, and it may remain active throughout the execution except for occasional suspensions for the purpose of synchronization with other objects. A *body* will be attached to each active object. Bodies of objects are functions that may be recursive and diverse (non-terminating). The active objects introduce coarse grain parallelism in PROOF:

P3. There can be a number of objects that are active simultaneously throughout the execution.

Synchronization of Objects

The methods defined in each object may require some preconditions under which the methods can be executed. This problem is commonly known as the synchronization problem. In order to specify the synchronization constraints, many constructs have been proposed. Among them are critical sections, message queues, behavior abstraction and enabled-sets. However, the use of these constructs interferes with the inheritance mechanism, and the synchronization constraints are not directly inheritable with the methods. In PROOF, the synchronization among the objects is achieved by attaching an optional precondition, called *guard*, to each of the methods in a class. Each guard is a predicate. The object which invokes the method is suspended when the attached guard evaluates to `False`, and it is resumed when the guard becomes `True`. The guard attached to a method is defined in a way that it depends only on the status of the local data, and does not depend on the definition of any other methods. Therefore, the inheritance of individual methods will not be hampered by

the inclusion of the guard. The guards can be inherited with the methods they are attached to.

By separating synchronization constraints (guard) from the behavior of methods (expression), integration of inheritance and parallelism can be achieved without interference. Since each method is associated with a guard specifying the condition under which the method can be executed, each method can be a natural unit of inheritance.

Persistence of Objects

A major deficiency of the functional paradigm is its history-insensitivity. PROOF is made history sensitive by making the objects persistent and allowing the reception of values by objects, i.e., the assignment of values to objects. The local data of objects is persistent. The reception of values by objects will modify the local data of objects. A pseudo-function $\mathcal{R}$, called the *reception function*, is introduced to denote the reception of a value by an object.

$$\mathcal{R} \llbracket o \rrbracket (e)$$

$\mathcal{R}$ is not a function, but can be treated as a function. $\mathcal{R}$ has two parameters: an object o, the recipient, and the expression e, to be received by o. e may contain applications of applicative functions only. This pseudo-function can appear only inside bodies of active objects, and may not be nested. Major differences between modification of objects through $\mathcal{R}$ and traditional assignments are

1. The evaluation of the expression e in $\mathcal{R}$ can be parallel, since e contains only applications of purely applicative functions.

2. No partial modification to the object o is allowed. The local data of an object can only be modified as a whole entity, i.e., its components cannot be modified individually.

The restriction that the reception function can appear only in the bodies of objects implies that all the methods in an object are still applicative, i.e., no side effect. This restriction effectively preserves the referential transparency at the method level and retains the parallelism owing to the referential transparency.

An object can simultaneously participate in more than one function evaluation. The potential for the simultaneous participation of an object in the evaluation of different functions also implies another source of coarse grain parallelism in PROOF:

P4. An object can be involved in two or more function evaluations simultaneously.

REFERENCES

[Adam74] Adam, T.L., Chandy, K.M. and Dickson, J.R. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM* 17, 12 (Dec. 1974), 685–690.

[America87] America, P. A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming.* Yonezawa, A. and Tokoro, M. (Eds.). MIT Press, Cambridge, MA (1987), 199–220.

[Babb84] Babb, R.G. Parallel Processing with Large Grain Data Flow Techniques. *IEEE Computer* (Jul. 1984), 55–61.

[Backus78] Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM* 21, 8 (Aug. 1978), 613–641.

[Bal89] Bal, H.E., Steiner, J.G. and Tanenbaum, A.S. Programming Languages for Distributed Computing Systems. *ACM Computing Survey* 21, 3 (Sep. 1989), 261–322.

[Bannister83] Bannister, J.A. and Trivedi, K.S. Task Allocation in Fault Tolerant Distributed Systems. *Acta Informatica* 20 (1983), 261–281.

[Bokhari87] Bokhari, S.H. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, Boston, MA 1987.

[Broy89] Broy, M. Requirement and Design Specification of Distributed Systems: The Lift Problem. In *Proceedings of Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Hong Kong (1988), 164–173.

[Casavant88] Casavant, T. and Kuhl, J. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering* (Feb. 1988), 141–154.

[Chatterjee91] Chatterjee, S., Blelloch, G.E. and Fisher, A.L. Size and Access Interface for Data-Parallel Programs. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto,Canada (Jun. 1991), 130–144.

[Chen90] Chen, D.K., Su, H.M. and Yew, P.C. The Impact of Synchronization and Granularity on Parallel Systems. *CSRD Report No. 942*, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL (1990).

[Chu80] Chu, W.W., Holloway, L.J., Lan, M.T. and Efe, K. Task Allocation in Distributed Data Processing. *IEEE Computer* 13, 11 (Nov. 1980), 57–69.

[Cvetanovic87] Cvetanovic, Z. The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems. *IEEE Transactions on Computers* 36, 4 (Apr. 1987), 421–432.

[Duncan90] Duncan, R. A Survey of Parallel Computer Architectures. *IEEE Computer* (Feb. 1990), 5–16.

[Efe82] Efe, K. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *IEEE Computer* 15, 6 (June 1982), 50–56.

[El-Rewini90] El-Rewini, H. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing* 9 (1990), 138–153.

[Fenton91] Fenton, W., Ramkumar, B., Saletore, V.A. and Sinha, A.B. Supporting Machine Independent Programming on Diverse Parallel Architecture. In *Proceedings of the 1991 International Conference on Parallel Processing Vol. II*, Austin, Texas (1991), 193–201.

[Gabber90] Gabber, E. VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 3 (Jul. 1990), 304–317.

[Gaudiot89] Gaudiot, J.L. and Lee, L.T. Occamflow: A Methodology for Programming Multiprocessor Systems. *Journal of Parallel and Distributed Computing* 7 (1989), 96–124.

[Ghezzi87] Ghezzi, C. and Jazayeri, M. *Programming Language Concepts.* John Wiley & Sons, New York, 1987.

[Graham69] Graham, R.L. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics* 17, 2 (Mar. 1969).

[Ha91] Ha, S. and Lee, E.A. Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration. *IEEE Transactions on Computers* 40, 11(Nov. 1991), 1225–1238.

[Huang85] Huang, J.P. Modeling of Software Partitioning for Distributed Real-Time Applications. *IEEE Transactions on Software Engineering* 11, 10 (Oct. 1985), 1113–1126.

[Hudak89] Hudak, P. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Survey* 21, 3 (Sep. 1989), 359–411.

[Hudak85] Hudak, P. and Goldberg, B. Distributed Execution of Functional Programs Using Serial Combinators. *IEEE Transactions on Computers* 34, 10 (Oct. 1985), 881–891.

[Hwang89] Hwang, J.J., Chow, Y.C., Anger, F.D. and Lee, C.Y. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM J. Comput.* 18, 2 (Apr. 1989), 244–257.

[Kasahara84] Kasahara, H. and Seinosuke, N. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers* 33, 11 (1984), 1023–1029.

[Keller84] Keller, R. and Lin, C. Simulated Performance of a Reduction-Based Multiprocessor. *IEEE Computer* (Jul. 1984), 70–81.

[Kruatrachue88] Kruatrachue, B. and Lewis, T. Grain Size Determination for Parallel Processing. *IEEE Software* (Jan. 1988), 23–32.

[Lee89] Lee, L.-T. OCCAMFLOW: Programming a Multiprocessor System in a High-Level Data-Flow Language, *Ph.D. dissertation*, University of Southern California, CA (Aug. 1989).

[Liberman87] Liberman, H. Concurrent Object-Oriented Programming in Act 1. In *Object-Oriented Concurrent Programming*, Yonezawa, A. and Tokoro, M. (Eds.). MIT Press, Cambridge, MA (1987), 9–36.

[Lo84] Lo, V.M. Heuristic Algorithms for Task Assignment in Distributed Systems. In *Proceedings of the 4th Int. Conference on Distributed Computing Systems* (May 1984), 30–39.

[Manarchi92] Manarchi, D.E. and Puhr, G.I. A Research Typology for Object-Oriented Analysis and Design. *Communications of ACM* 35, 9 (Sep. 1992), 35–47.

[Mann90] Mannino, M.V. and Choi, I.J. The Object-Oriented Functional Data Language. *IEEE Transactions on Software Engineering* 16, 11 (Nov. 1990), 1258–1272.

[McCreary89] McCreary, C. and Gill, H. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM* (September 1989), 1073–1078.

[McGraw85] McGraw, J. and Skedzielewski, S. SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. *Technical Report M-146*, Lawrence Livermore National Laboratory, Livermore, CA (Mar. 1985).

[Milner84] Milner, R. A Proposal for Standard ML. In *Proceedings of 1984 ACM Conference on LISP and Functional Programming* (1984), 184–197.

[Peyton87] Peyton Jones, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ 1987.

[Poly87] Polychronopoulos, C.D. and Banerjee, U. Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds. *IEEE Transactions on Computers* 4, 36 (1987), 410–420.

[Quinn90] Quinn, M.J. and Hatcher, P.J. Data-Parallel Programming on Multicomputers. *IEEE Software* (Sep. 1990), 69–76.

[Roman87] Roman, G. Specifying Software/Hardware Interactions in Distributed Systems. In *Proceedings of the International Conference on Software Engineering* (1987), 126–139.

[Sarkar87] Sarkar, V. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. *Ph.D. dissertation*, Stanford University, CA (Apr. 1987).

[Sarkar88] Sarkar, V., Skedzielewsi, S. and Miller, P. An Automatically Partitioning Compiler for SISAL. *Technical Report UCRL-98289*, Lawrence Livermore National Laboratory, Livermore, CA (Dec. 1988).

[Schwan85] Schwan, K. *Tailoring Software for Multiple Processor Systems.* UMI Research Press, Ann Arbor, MI (1985).

[Shatz88] Shatz, S.M. and Wang, J.P. *Distributed Software Engineering.* IEEE Computer Society Press, Washington, D.C., 1988.

[Shatz86] Shatz, S.M. and Yau, S.S. A Partitioning Algorithm for Distributed Software Systems Design. *Information Science* 38 (1986), 165–180.

[Shen85] Shen, C.C. and Tsai, W.H. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criteria. *IEEE Transactions on Computers* 34, 3 (Mar. 1985), 197–203.

[Stone77] Stone, H.S. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering* 3, 1 (Jan. 1977), 85–93.

[Towley86] Towley, D.F. Allocating Programs Containing Branches and Loops within a Multiple Processor System. *IEEE Transactions on Software Engineering* 12 (Oct. 1986), 1018–1024.

[Turner90] Turner, K. A Lotos-Based Development Strategy. In *Formal Description Techniques, II.* Vuong, S. (Ed.). North-Holland, Amsterdam (1989), 117–132.

[Ullman75] Ullman, J. NP-complete Scheduling problems. *J. Compt. System Science* 10 (1975), 384–393.

[Vissers90] Vissers, C., Lagemaat, J. and Pires, L. Formal Description Techniques for Distributed Computing Systems, the Challenge for the 1990's. In *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems*, Cairo, Egypt (1990), 456–471.

[Weber86] Weber, H. and Ehrig, H. Specification of Modular Systems. *IEEE Transactions on Software Engineering* 12, 7 (July 1986), 784–798.

[Weber88] Weber, H. and Ehrig, H. Specification of Concurrently Executable Modules and Distributed Modular Systems. In *Proceedings of Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Hong Kong (1988), 202–215.

[Wall91] Wall, D. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), 59–70.

[Wikstrom88] Wikstrom, A. *Standard ML.* Prentice-Hall, Englewood Cliffs, NJ 1988.

[Yau90] Yau, S. S., Jia, X. and Bae, D.-H. Trends in Software Design for Distributed Computing Systems. In *Proceedings of the Second IEEE Workshop on the Future Trends of Distributed Computing Systems*, Cairo, Egypt (1990), 154–160.

[Yau91] Yau, S. S., Jia, X. and Bae, D.-H. PROOF: A Parallel Object-Oriented Functional Computation Model. *Journal of Parallel and Distributed Computing* 12 (1991), 202-212.

[Yau91b] Yau, S. S., Jia, X., Bae, D.-H.,Chidambaram, M. and Oh, G. An Object-Oriented Approach to Software Development for Parallel Processing Systems. In *Proceedings of 15th Annual Int'l Computer Software & Applications Conf. (COMPSAC91)* (1991), 453–458.

[Yau92] Yau, S.S., Bae, D.-H. and Chidambaram, M. A Framework for Software Development for Distributed Parallel Computing Systems. In *Proceedings of the Third Workshop on the Future Trends of Distributed Computing Systems*, Taipei, Taiwan (1992), 240–246.

[Yau91c] Yau, S. S. and Wiharja, I. An Approach to Module Distribution for the Design of Embedded Distributed Software Systems. *Information Science* 56, (1991), 1–22.

[Yau81] Yau, S. S., Yang, C. C. and Shatz, S. M. An Approach to Distributed Computing System Software Design. *IEEE Transactions on Software Engineering* 7, 4 (Jul. 1981), 427–436.

[Yokote87] Yokote, Y. and Tokoro, M. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming.* Yonezawa, A. and Tokoro, M. (Eds.). MIT Press, Cambridge, MA (1987), 129–158.

BIOGRAPHICAL SKETCH

Ruey-Ming Yang was born in Taipei, Taiwan. In 1975 he received a B.S. in computer science and control engineering from National Chiao Tung University. In 1977 he received a MBA from the Institute of Management, National Chiao Tung University. During his military service from 1977 to 1979 he served in the College of National Defense Management as an instructor for courses in the area of computer science and management science. From 1979 to 1980 he worked for Taiwan International Standard Electronics Corporation (an Affiliate of ITT) as a system analyst. Since 1980 he has been with the Institute for Information Industry (III) in Taiwan, ROC, as an engineer, project manager, and the director of Planning and Evaluation Office. He entered the Ph.D. program of the Computer and Information Sciences Department at the University of Florida in the Fall of 1989.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stephen S. Yau, Chairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Li-Min Fu
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

5-III-93

Nabil Kamel
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Chung-Yee Lee
Associate Professor of
Industrial and Systems Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Yann-Hang Lee
Associate Professor of
Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May, 1993

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School